Order Number 9215910

Asynchronous algorithms for shared memory machines

Wu, Michael M., Ph.D.

University of Illinois at Urbana-Champaign, 1992

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

ASYNCHRONOUS ALGORITHMS FOR SHARED MEMORY MACHINES

BY

MICHAEL M. WU

B.S., University of Illinois, 1985
M.S., University of Illinois, 1987

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1992

Urbana, Illinois

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

THE GRADUATE COLLEGE

NOVEMBER 1991

WE HEREBY RECOMMEND THAT THE THESIS BY

MICHAEL M. WU

ENTITLED ASYNCHRONOUS ALGORITHMS

FOR SHARED MEMORY MACHINES

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF DOCTOR OF PHILOSOPHY

Michael C. Loui
Director of Thesis Research

Timothy M. Pinkston
Head of Department

Committee on Final Examination†

Michael C. Loui Chairperson

Bruce Hajek

Christos Papadimitriou

W. Kent Fuchs

† Required for doctor's degree but not for master's.

O 517

ABSTRACT

In an effort to develop more realistic models of computation, we introduce several asynchronous shared memory machines and design asynchronous algorithms for those machines. We first model asynchronous protocols for communication across unreliable channels using finite-state machines communicating via an unreliable shared memory. We establish lower bounds on the size of machines and the number of symbols in the transmission alphabet required to achieve reliable communication. We consider two types of finite-state machines and two fault models for the shared memory. In each case, we show that there are robust protocols for deletion and insertion errors. We also show that there are no robust protocols for mutation errors. In contrast, in the synchronous case, robust protocols exist for all of these types of errors.

The Parallel Random Access Machine (PRAM) is a fundamental model of parallel computation, but it is not physically realizable. We introduce a more realistic model of parallel computation, the Asynchronous PRAM (APRAM). Let G be a graph with n vertices and m edges. We present two APRAM models and algorithms to find the connected components of G for each model. Algorithm I runs on an APRAM with only atomic **read** and **write** primitives and requires $O(n \log n)$ rounds. Algorithms II and III run on an APRAM with limited **read-modify-write** primitives and require $O(\log n)$ rounds. Algorithm III is more efficient than Algorithm II and requires fewer global synchronizations. All three algorithms use $m + n$ processors. We then modify our APRAM connected components algorithms to obtain APRAM algorithms for finding a spanning forest or a minimum spanning forest of G.

Finally, we present an APRAM algorithm for finding the biconnected components of a connected graph G. Our biconnected components algorithm runs on an APRAM with limited **read-modify-write** primitives and requires $O(\log n)$ rounds using $O(m + n)$ processors.

ACKNOWLEDGMENTS

First, I would like to thank my advisor Professor Michael Loui for his guidance, insight, and steadfast encouragement during the course of my graduate studies over the past six years. His dedication and enthusiasm are second to none. I would also like to acknowledge the other members of my doctoral committee: Professors Kent Fuchs, Bruce Hajek, and Constantine Polychronopoulos. I also thank Professor Franco Preparata for serving on my preliminary examination committee.

I am grateful to my family and friends for providing moral support and the welcome distractions. In particular, I thank Mary Allison, Tim Yao, and Lyle Kipp. Finally, I would like to thank Hosame Abu-Amara, Jerry Trahan, Marsha Woerner, David Luginbuhl, Nancy Amato, David Atkinson, Pat McGuinness, and the other members of my research group.

The following sources gave financial support for the research in this thesis: the University of Illinois Campus Research Board, Kodak through a New Faculty Incentive Grant, the Office of Naval Research under Contract N00014-85-K-0570, and the Joint Services Electronics Program under Contract N00014-90-J-1270.

TABLE OF CONTENTS

CHAPTER	PAGE
1. INTRODUCTION	1
2. MODELING ROBUST ASYNCHRONOUS COMMUNICATION PROTOCOLS WITH FINITE-STATE MACHINES	6
2.1. Introduction	6
2.2. The System Model	9
2.3. Complexity of Communicating Finite-state Machines	12
2.4. Transmission in the Presence of Deletion Errors	19
2.4.1. Compulsive machines and read faults	19
2.4.2. Selective machines and read faults	24
2.4.3. Selective machines and write faults	25
2.4.4. Compulsive machines and write faults	26
2.5. Transmission in the Presence of Insertion Errors	30
2.6. Transmission in the Presence of Mutation and Multiple Errors	32
2.7. Conclusions	33
3. PARALLEL ALGORITHMS FOR MINIMUM SPANNING TREE IN LOGARITHMIC TIME WITHOUT PRIORITY WRITING	37
3.1. Introduction	37
3.2. Model of Computation	39
3.3. The MST Algorithm of Awerbuch and Shiloach	39
3.4. Common CRCW PRAM MST Algorithm	43
4. ASYNCHRONOUS PARALLEL ALGORITHMS FOR GRAPH CONNECTIVITY AND RELATED PROBLEMS	47
4.1. Introduction	47
4.2. Model of Computation	51
4.2.1. The PRAM	51
4.2.2. The asynchronous PRAM	52
4.3. The Design of APRAM Algorithms	54
4.3.1. Wait-free data structures	55
4.3.2. Processor synchronization	56
4.4. Synchronous Algorithms	59
4.4.1. Connected components algorithms	59
4.4.2. Spanning forest algorithm	63
4.4.3. Minimum spanning forest algorithm	64
4.5. APRAM Connected Components Algorithm I	65

4.6. APRAM Connected Components Algorithm II 72
4.6.1. The algorithm 72
4.6.2. Analysis 75
4.6.2.1. The potential function 77
4.6.2.2. Contribution 83
4.6.2.3. Performance 85
4.7. APRAM Connected Components Algorithm III 104
4.8. APRAM Spanning Forest Algorithms 111
4.8.1. Spanning forest algorithms 111
4.8.2. Minimum spanning forest algorithms 112
4.9. Conclusions 114

5. AN ASYNCHRONOUS PARALLEL ALGORITHM FOR COMPUTING BICONNECTED COMPONENTS 115
5.1. Introduction 115
5.2. Model of Computation 116
5.3. The Synchronous Biconnected Components Algorithm 117
5.4. APRAM Biconnected Components Algorithm 123

REFERENCES 126

VITA 132

LIST OF TABLES

Table Page

2.1. Existence of robust protocols in the presence of various errors 35

4.1. A summary of the definitions of stage, step, round, and phase 70

LIST OF FIGURES

Figure Page

2.1. A robust protocol when there are no transmission errors 18

2.2. A robust protocol for deletion errors for compulsive machines 20

2.3. A robust protocol for deletion errors for selective machines and write faults..... 25

2.4. A robust protocol for insertion errors for compulsive machines and write faults 32

2.5. A robust protocol for deletion and insertion errors for selective machines and write faults..... 34

4.1. An example of a forest F 80

4.2. An example of a cluster 84

CHAPTER 1.

INTRODUCTION

A parallel computer is a machine which consists of two or more processors, each with its own local memory, and in addition, possibly some globally shared memory. One well-studied model of a parallel computer is the Parallel Random Access Machine (PRAM). The PRAM is an idealized parallel computer that allows algorithm designers to concentrate on the computational aspects of problems without concern about the reliability and synchronization of processors.

Although the PRAM is an important and fundamental model of parallel computation, it is not physically realizable. In designing algorithms for real machines, the PRAM hides the cost of synchronization. As the number of processors becomes large, it becomes impractical to synchronize the processors using a single global clock. Synchronization mechanisms, whether implemented through software or hardware, slow the execution of algorithms. In addition, to prevent more than one processor from simultaneously accessing the same cell in the shared memory, read and write operations on the shared memory must also be synchronized.

In an effort to develop more realistic models of computation, in this thesis we introduce several asynchronous shared memory machines and design asynchronous algorithms for those machines. In the remainder of this chapter, we give an overview of the thesis.

In Chapter 2, we study asynchronous data-link communication protocols. Consider a data communications system consisting of a transmitter and a receiver communicating across

unreliable channels. The goal of the system is for the transmitter to communicate its input, an infinite sequence of 0's and 1's, to the receiver, and for the receiver to output this sequence of bits.

Aho et al. (1982) used communicating finite-state machines to model synchronous protocols for reliable communication across unreliable channels. We extend their ideas and model asynchronous protocols for communication across unreliable channels using finite-state machines communicating via an unreliable shared memory. We establish lower bounds on the size of machines and the number of symbols in the transmission alphabet required to achieve reliable communication.

We consider two types of finite-state machines, compulsive machines and selective machines. In the compulsive model, a machine sends a symbol at every transition. In the selective model, a machine does not have to send a symbol at every transition. We also consider two fault models for the shared memory. In the read fault model, write operations are reliable but read operations are unreliable. In the write fault model, read operations are reliable but write operations are unreliable. These seemingly minor differences in the system model can significantly affect the results, and thus these differences illustrate the need for careful definitions of the machine and shared memory models.

For each combination of finite-state machine and memory fault model, we show that there are robust protocols for deletion and insertion errors. We present protocols for compulsive machines in the read fault model and for selective machines in both the read and write fault models that have the minimum number of states for the machines and the minimum number of symbols in the transmission alphabet simultaneously. For compulsive machines in the write fault model, our protocol uses the minimum number of symbols when

the transmitter and receiver have only two states each. We also show that there are no robust protocols for mutation errors. In contrast, in the synchronous case, robust protocols exist for all of these types of errors. Chapter 2 has been accepted for publication in *IEEE Transactions on Communications* (Wu and Loui, 1991).

In Chapter 3, we present Common Concurrent Read Concurrent Write (CRCW) PRAM algorithms for finding the minimum spanning tree (MST) of a graph. Although the algorithms of Chapter 3 are synchronous, the design of these algorithms provides insight into the design of the asynchronous algorithms of Chapter 4.

Let G be a graph with n vertices and m edges. Hirschberg (1982) gave a randomized Common CRCW PRAM algorithm for finding the MST of G that runs in $O(\log n)$ expected time using n^3 processors. Awerbuch and Shiloach (1987) gave a MST algorithm for the Priority CRCW PRAM that runs in $O(\log n)$ time using $O(m + n)$ processors.

We present a deterministic algorithm for finding the MST of G that runs on a Common CRCW PRAM in $O(\log n)$ time using $2m + n^{1+2\varepsilon}$ processors, where ε is a constant such that $0 < \varepsilon \leq 1/2$. Our algorithm has the same running time as the algorithm of Hirschberg, but is deterministic and uses fewer processors. For mildly dense graphs, where $m = \Omega(n^{1+2\varepsilon})$, our algorithm has the same performance as the algorithm of Awerbuch and Shiloach using a weaker CRCW PRAM model. The algorithms of Chapter 3 have been published in a technical report (Wu, 1990).

In Chapter 4, we present a more realistic model of parallel computation, the Asynchronous PRAM (APRAM). We introduce two APRAM models and design APRAM algorithms to find the connected components of G for each model.

We show that oblivious PRAM algorithms can be easily converted into APRAM algorithms. The connected components of G can be found obliviously by computing the transitive closure of the adjacency matrix of G. This method is inefficient, however. The fastest known CRCW PRAM algorithm for computing the transitive closure of an $n \times n$ Boolean matrix requires $O(\log^2 n)$ time using $O(n^{2.376})$ processors (Karp and Ramachandran, 1990).

Nonoblivious CRCW PRAM algorithms can find the connected components of G much more efficiently. But it is also more difficult to implement nonoblivious PRAM algorithms on an APRAM efficiently. Although several researchers have developed APRAM algorithms, with only one exception, all of the algorithms have been oblivious.

All three of our connected components algorithms are nonoblivious and require significantly fewer global synchronizations than a straightforward asynchronous implementation of a synchronous algorithm. Algorithm I runs on an APRAM with only atomic **read** and **write** primitives and requires $O(n \log n)$ rounds. Algorithms II and III run on an APRAM with limited **read-modify-write** primitives, specifically **replace-min** and **increment**, and require $O(\log n)$ rounds. Algorithm III is more efficient than Algorithm II and uses fewer global synchronizations. All three algorithms use $m + n$ processors. Finally, we modify our APRAM connected components algorithms to obtain APRAM algorithms for finding a spanning forest and a minimum spanning forest of G.

In Chapter 5, we use the algorithms of Chapter 4 to design an APRAM algorithm for finding the biconnected components of G. The algorithm runs on an APRAM with **replace-min** and **increment** primitives and requires $O(\log n)$ rounds using $m + n$ processors.

We discuss the related results and review the literature for each of the various problems in their corresponding chapters. Each chapter is self-contained.

CHAPTER 2.

MODELING ROBUST ASYNCHRONOUS COMMUNICATION PROTOCOLS WITH FINITE-STATE MACHINES

2.1. Introduction

Consider a data communications system consisting of a transmitter and a receiver communicating across unreliable channels. The transmitter has a channel to send symbols to the receiver, and the receiver has a separate channel to send symbols to the transmitter. The goal of the system is for the transmitter to communicate its input, an infinite sequence of **0**'s and **1**'s, to the receiver, and for the receiver to output this sequence of bits.

Let the transmission alphabet Σ be a finite set of symbols including the null symbol ε, and nonnull symbols **0**, **1**, **2**, **A**, **B**, **C**, $\cdots$. The transmitter and receiver send symbols of Σ to each other to communicate the input of the transmitter to the receiver. Since the channels are unreliable, the symbols sent between the transmitter and receiver may be corrupted. We consider the following kinds of errors in channels:

(1) **Deletion:** a nonnull symbol is sent and ε is received.

(2) **Mutation:** a nonnull symbol is sent and a different nonnull symbol is received.

(3) **Insertion:** ε is sent and some nonnull symbol is received.

Communication protocols are used to ensure reliable transmission of data across unreliable channels. A protocol is *robust* under a certain class of errors if in the presence of those errors the protocol satisfies the following conditions under all possible executions:

(1) **Safety:** The output sequence is always a prefix of the input sequence.

(2) **Liveness:** If the receiver has not output the complete input sequence and if no transmission error occurs within some finite number of alternations of steps by the transmitter and receiver, then the receiver outputs at least one more bit.

Communicating finite-state machines have been used to model and validate communication protocols. Bartlett et al. (1969) used finite-state machines to model a protocol for reliable full-duplex transmission over half-duplex links. Danthine (1982) used finite-state machines to model computer network protocols.

Zafiropulo et al. (1980) and Brand and Zafiropulo (1983) used communicating finite-state machines to investigate the problem of protocol verification. Yu and Gouda (1982) presented an algorithm to determine whether two communicating finite-state machines may deadlock. Peng and Purushothaman (1989) presented a dataflow approach to analyze protocols for deadlock and unspecified reception.

Gouda and The (1985) used communicating finite-state machines to model two protocols for the physical layer of the International Standards Organization (ISO) reference model for computer networks (Tanenbaum, 1981). The first was an asynchronous start-stop protocol and the second was a protocol for synchronous transmission with modems. They gave a methodology to verify communication boundedness and progress for each protocol. Lynch et al. (1988) used input/output automata to specify the physical and data link layers formally. They showed that no data link layer protocol can tolerate crashes of the host processors on which the protocol runs.

For faulty asynchronous communications channels where messages can be reordered or deleted, Wang and Zuck (1989) used a knowledge-based approach to derive tight bounds on

the number of different sequences that can be transmitted when the data items and the message alphabet have finite domains. For the same model, Temporo and Ladner (1990) introduced three characteristics of protocols, including *transmission cost*, *recovery cost*, and *lookahead*, and gave tight bounds on the efficiency of protocols in terms of these measures.

Aho et al. (1982) used finite-state machines to model synchronous protocols for data communication and derived lower bounds on the size of machines needed to achieve reliable communication across unreliable channels. Several researchers have proved the correctness of their protocols. Hailpern (1985) used the finite-state machine and abstract-program approaches, Gouda (1985) used reachability graphs, and Halpern and Zuck (1987) used a knowledge-based approach.

In this chapter, we extend the ideas of Aho et al. and consider finite-state machines communicating asynchronously across unreliable channels. We establish lower bounds on the size of machines and the number of symbols in the transmission alphabet needed to achieve reliable communication. Aho et al. showed that in the synchronous case there are robust protocols if at most two of deletion, mutation, and insertion errors are present. We show that in the asynchronous case there are robust protocols if deletion and insertion errors are present, but no robust protocols if mutation errors are present.

Our results illustrate the difference between synchronous communication and asynchronous communication. Furthermore, we distinguish between read faults and write faults, and between two kinds of machines. We consider the identification of these distinctions to be the other major contribution of this chapter, for these distinctions emphasize the need for careful definitions of fault models and machines.

In Section 2.2, we describe our models of the communication system. In Section 2.3, we establish lower bounds on the size of machines and the number of symbols in the transmission alphabet required to achieve robust communication. In Section 2.4, we consider transmission in the presence of deletion errors. In Section 2.5, we consider insertion errors. In Section 2.6, we consider mutation and multiple types of errors. We summarize our results in Section 2.7.

2.2. The System Model

The data communications system abstracts the physical and data link layers of the ISO computer network model. The physical layer is concerned with transmitting raw bits over a communication channel between processors without regard for transmission errors. The purpose of the data link layer is to make the physical layer appear free of transmission errors to the network layer. Thus the input and output of the data communications system are the bits that are sent from one processor to another processor at the data link layer.

In our system model, processors communicate asynchronously. Lamport (1986) discussed the nature of asynchronous communication. Since the sender does not know whether the receiver is observing the communication medium while the sender is modifying it, any change the sender makes to the state of the communication medium must remain after the sender has finished its transmission so that the receiver can sense the change at a later time. Lamport calls this a *persistent* communication act.

The transmitter and receiver are modeled by finite-state machines. Let T be the transmitter and let R be the receiver. Machines T and R communicate by sending symbols of Σ to each other. Transmitter T has an unreliable channel to send symbols to R, and R has

a separate unreliable channel to send symbols to T. On each channel, we will refer to the sending machine as the *source* and the receiving machine as the *destination.*

The unreliable channels are modeled by an unreliable shared memory with read and write operations. The memory simulates the persistent signals of a physical system. We consider two fault models for the shared memory. In the *read fault* model, write operations are reliable but read operations are unreliable. The symbol that the destination reads from a memory cell c might not be the same as the symbol stored in c. The various transmission errors are as follows:

(1) A deletion error occurs if a nonnull symbol is stored in c and the destination reads an ϵ.

(2) A mutation error occurs if a nonnull symbol is stored in c and the destination reads a different nonnull symbol.

(3) An insertion error occurs if ϵ is stored in c and the destination reads a nonnull symbol.

In the *write fault* model, read operations are reliable but write operations are unreliable. The symbol that is stored in a memory cell c might not be the same as the symbol that the source writes into c. Since the channels are unidirectional, the source is not permitted to read c to determine whether a write operation is correct. The various transmission errors are as follows:

(1) A deletion error occurs if the source writes a nonnull symbol into c and an ϵ is stored.

(2) A mutation error occurs if the source writes a nonnull symbol into c and a different nonnull symbol is stored.

(3) An insertion error occurs if the source writes an ϵ into c and a nonnull symbol is stored.

The read fault model corresponds to the case in which errors may occur only within the communication medium or when a machine is receiving a signal. The write fault model corresponds to the case in which errors may occur only when a machine is sending a signal. We consider two fault models because if the location and types of errors that can occur are limited, then we can derive optimal protocols in some cases.

Machines T and R have two shared memory cells that represent the two unreliable channels. Each machine has one *outgoing* cell and one *incoming* cell. Let c_T be T's outgoing cell and c_R be R's outgoing cell. Transmitter T's outgoing cell is R's incoming cell and vice versa. A machine *sends* a symbol a by writing a into its outgoing cell. A machine *receives* a symbol by reading its incoming cell. The value of a cell may change only when it is written. Thus, a read operation does not change the value in a cell. Initially, both cells contain ϵ.

Transmitter T has an *input buffer* from which it reads input bits and an *input pointer* to keep track of the current input bit. Receiver R has an *output buffer* into which it writes output bits. During a move, a machine may perform a *buffer operation.* The buffer operation for T is advancing its input pointer to the next bit in the input buffer. The buffer operation for R is writing a bit into the output buffer. For convenience, we assume that the input is infinite. Thus we will not need to specify the termination of protocols at the end of a finite input.

We consider two types of finite-state machines for T and R. In the *compulsive* model, a machine sends a symbol of Σ at every transition. A move of T depends on its current state, the current bit in the input buffer, and the current symbol in c_R. In one move T changes state, sends a symbol to R, and chooses whether to advance its input pointer to the next bit in the

input buffer. A move of R depends on its current state and the current symbol in c_T. In one move R changes state, sends a symbol to T, and chooses whether to write a bit into the output buffer. A *state transition* $\delta_R(s,a)=s'$ for R means that if the current state of R is s and R reads a from its incoming cell, then R makes a transition to state s'. If the states s and s' are the same, then the transition is a *self-loop*. A *transition output* $\lambda_R(s,a)=b$ means that on the state transition $\delta_R(s,a)$ machine R writes b into its outgoing cell.

In the *selective* model, a machine does not have to send a symbol at every transition. In one move a machine changes state, chooses whether to send a symbol, and chooses whether to perform a buffer operation. If a machine does not send a symbol on some state transition, then there is no corresponding transition output. In Section 2.4.4, we will show that the selective and compulsive models are different. We show that in the presence of deletion errors and write faults we can design more efficient protocols for the selective model than for the compulsive model.

With both the compulsive and selective models of finite-state machines, the system is *asynchronous*. Thus the moves of T and R are arbitrarily interleaved. We assume, however, that each move of a machine is atomic. This ensures that if a machine sends a symbol during a move, then the symbol it sends is based on the current symbol in its incoming cell.

2.3. Complexity of Communicating Finite-state Machines

In this section, we establish lower bounds on the size of machines and the number of symbols in the transmission alphabet needed to achieve reliable communication. These results apply to both compulsive and selective finite-state machines. We first consider the size of machines. As in Aho et al., we use the number of states as the measure of complexity.

This is a fair measure because the number of states is closely related to the size of the circuits needed to implement the machines.

In the synchronous case, if there are no transmission errors, then there is a simple robust protocol consisting of a one-state transmitter and a one-state receiver: the transmitter sends the next bit to the receiver after it receives an acknowledgment for the previous bit. Aho et al. showed that in the presence of deletion errors there are no robust one-state transmitters or receivers. We show that in the asynchronous case there are no robust one-state transmitters or receivers, even if there are no transmission errors, regardless of the size of the transmission alphabet.

Theorem 2.3.1: There is no robust one-state transmitter.

Proof: If T has one state, then the moves of T are self-loops and depend only on the current bit in the input buffer and the current symbol in c_R. Consider the execution of T on the input consisting of k **0**'s followed by a **1**, where $k \geq 2$. Suppose T's next move is the move on which T advances its input pointer from the first **0** to the second **0**. When T reads c_R, T advances its input pointer to the second **0**. Since T and R are asynchronous, T may read c_R an arbitrary number of times before R writes a new symbol.

Suppose T reads c_R twice before R moves. Then T advances its input pointer twice and skips a **0**. The execution of R in this case is the same as an execution of R on an input sequence of $k-1$ **0**'s followed by a **1** where T does not skip any **0**'s. Thus R outputs a sequence of $k-1$ **0**'s followed by a **1**, violating the safety condition of a robust protocol. □

Theorem 2.3.2: There is no robust one-state receiver.

Proof: The proof is similar to the proof of Theorem 2.3.1. If R has one state, then the moves of R are self-loops and depend only on the current symbol in c_T. Consider the execution of

R on an input of **01** to T. Suppose R's next move is the move on which R writes **0** into its output buffer. Since R and T are asynchronous, R may read c_T an arbitrary number of times and output an arbitrary number of **0**'s before T writes a new symbol. Thus the output may violate the safety condition of a robust protocol. □

Next we consider the number of symbols required in the transmission alphabet to achieve robust communication. We will show that in the asynchronous case at least three symbols are necessary, even if there are no transmission errors, no matter how many states the machines have. In contrast, in the synchronous case there are robust protocols that use only two symbols.

Let T and R be the machines of a robust protocol. Fix $a \in \Sigma$. An *a-sequence* of R is a sequence of distinct states $s_0, s_1, \ldots, s_k$ such that $\delta_R(s_i, a) = s_{i+1}$ for $i = 0, \ldots, k-1$. An *a-cycle* of R is a sequence of distinct states $s_0, s_1, \ldots, s_k$ such that states $s_0, s_1, \ldots, s_k$ form an a-sequence and $\delta_R(s_k, a) = s_0$. A self-loop is an a-cycle with one state.

Lemma 2.3.3: R does not write into the output buffer in the transitions of an a-cycle.

Proof: Suppose to the contrary that R does write into the output buffer in an a-cycle. Since T and R are asynchronous, R may read c_T an arbitrary number of times before T writes a new value. If R is in a state of an a-cycle and a is in c_T, then R may make an arbitrary number of transitions in the a-cycle and output an arbitrary number of bits. Thus the output may violate the safety condition. □

A machine is *reduced* if for all a, the only a-cycles are self-loops. A sequence of transitions on a is *maximal* if the last transition of the sequence is a self-loop on a.

Lemma 2.3.4: Suppose R is not a reduced machine. Then there is a reduced machine R' such that if R is replaced with R', then the protocol with T and R' is robust.

Proof: To obtain R', we break all the a-cycles of R with more than one state. Let $s_0, s_1, \ldots, s_k$ be an a-cycle of R. Let s_i be a state of the a-cycle such that there is a transition from s_i on b, where $b \neq a$, that leads to a sequence of transitions where R writes into the output buffer without first reentering a state of the a-cycle. Let $b_{i-1} = \lambda_R(s_{i-1}, a)$, where $i-1$ is interpreted mod $k+1$. To break the a-cycle, replace the transition $\delta_R(s_i, a)$ and its output, if any, with the self-loop $\delta_{R'}(s_i, a) = s_i$ and make the transition output $\lambda_{R'}(s_i, a) = b_{i-1}$. By Lemma 2.3.3, R does not write into the output buffer during this transition. In this fashion, we break every a-cycle and construct a machine R' whose only a-cycles are self-loops, for all a.

Note that in breaking an a-cycle we cannot choose an arbitrary s_i. Suppose we choose a state s_i of R such that all sequences of transitions from s_i return to some state of the a-cycle and reenter state s_i without writing into the output buffer. If R' enters state s_i, then R' cannot output any more bits.

Next, we show that if R is replaced with R', then the protocol with T and R' is robust. Suppose R' is in state s_0 of the a-cycle and a is in c_T. If R' reads its incoming cell $j \leq i$ times before T makes its next move, then R enters state s_j and writes b_{j-1} into c_R. Thus, R' enters the same state and sends the same symbol to T as R would have.

If R' reads its incoming cell more than i times before T makes its next move, then R' arrives at state s_i and writes b_{i-1} into c_R. In this case, the execution of T and R' would be the same as an execution of T and R where R read its incoming cell i times before T makes its next move. Thus R' leaves state s_i by making the same transition R would have. □

By Lemma 2.3.4, we may assume without loss of generality that R is a reduced machine. We now establish a lower bound on the number of symbols required in the transmission alphabet. Note that the proof is independent of the number of states and does not depend on the finiteness of the state set.

Theorem 2.3.5: Any robust protocol requires at least three symbols in its transmission alphabet.

Proof: Suppose to the contrary that two symbols **A** and **B** suffice. Let $\alpha \in \{\mathbf{A},\mathbf{B}\}$. Intuitively, since in the asynchronous case the repetition of a symbol is indistinguishable, there are essentially only two unbounded sequences of symbols that can be sent from T to R. Thus there are not enough histories to encode all of the possible inputs.

More precisely, consider an execution in which whenever T writes a symbol α, R makes a maximal sequence of transitions on α before T writes a new symbol. Since there are only two symbols, each time T writes a new symbol, R alternately leaves states with a self-loop on **A** and states with a self-loop on **B**. Thus, the sequence of transitions R makes between different states is fixed after the first transition.

By Lemma 2.3.3, R does not output any bits on moves that are self-loops; otherwise R may output arbitrarily many bits. Thus R may output a bit only if a move of R causes R to enter a different state. But since the sequence of transitions made by R between different states is fixed after the first, so is the sequence of buffer operations made by R. Thus the output of R is a predetermined sequence of bits. If σ and σ' are two output strings of R and $|\sigma| < |\sigma'|$, then σ is a prefix of σ'. We call this the *prefix property.*

It is clear that there are input strings to T that begin with the same bit but do not have the prefix property. Thus there are input strings to T that cannot be output by R. It follows

then that there is no robust protocol that uses only two symbols in its transmission alphabet. □

We note that the result of Theorem 2.3.5 can be used to establish the need for a strobe or "data ready" signal in asynchronous hardware designs since data lines carry only two signals. One example of its use is for asynchronous data transfer between a CPU and I/O interfaces in computer systems (Mano, 1982).

We now present a simple asynchronous protocol in Figure 2.1 for data communication in the case in which there are no transmission errors. Other protocols in later sections are variations of this protocol. For clarity, we give names to the symbols. An arc from state s to state s' labeled a / b means that if the machine is in state s and reads a from its incoming cell, then it changes state to s' and writes b into its outgoing cell. The asterisk (*) indicates that T advances its input pointer, and the dagger (†) indicates that R writes a bit into its output buffer.

The initial states of T and R are p and r, respectively. Initially, both shared memory cells contain ε. When T is in state p, T tells R that T is ready to transmit the current bit in its input buffer by sending SYNC to R. When R is in state r, R is ready to request the next output bit from T. When R receives SYNC, R sends REQ to T and moves to state s. When T receives REQ, T sends DATA to R, where DATA represents the current bit in T's input buffer, and moves to state q. When R receives DATA, R writes the bit corresponding to DATA into its output buffer, sends ACK to T, and moves to state r. When T receives ACK, T advances its input pointer, sends SYNC to R, and moves to state p. Then the cycle repeats.

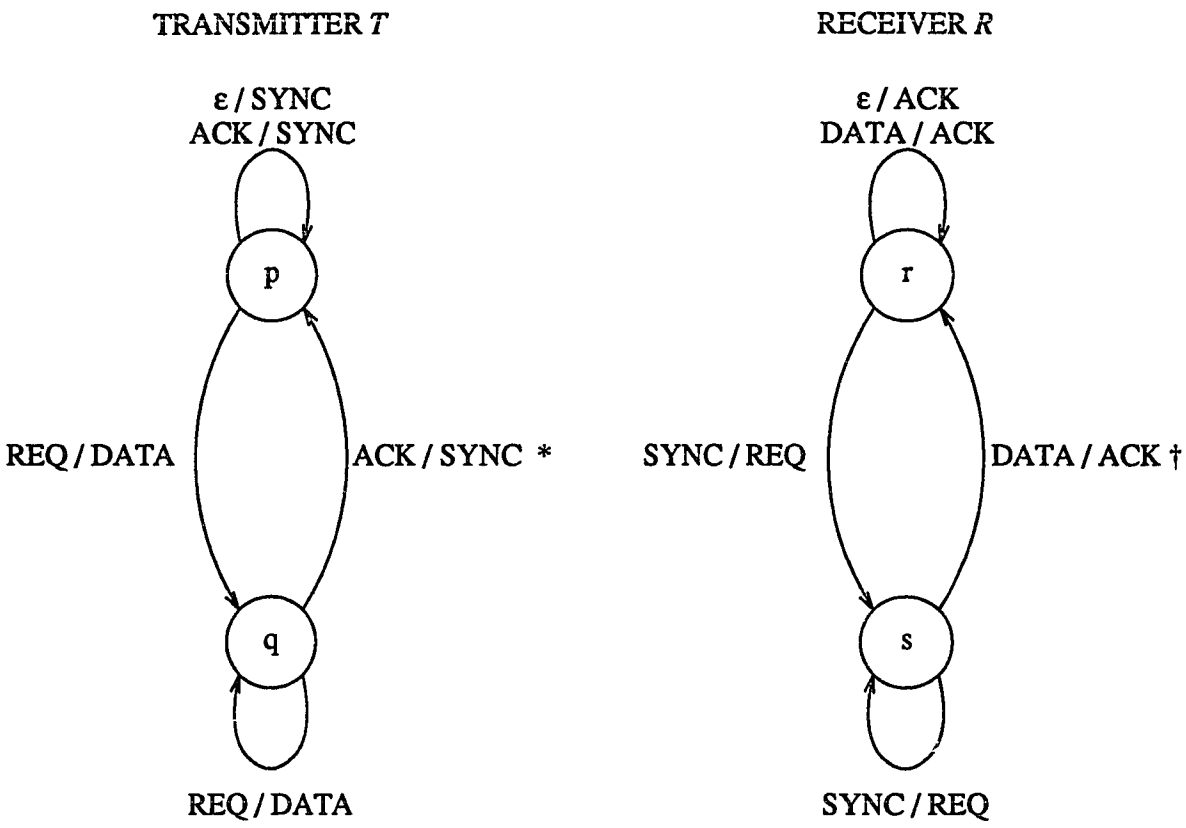

Figure 2.1. A robust protocol when there are no transmission errors.

Note that T and R alternately move to new states. Thus, during one cycle, T communicates one input bit to R, which R writes into its output buffer, and T advances its input pointer one bit. It is clear that the protocol satisfies the safety and liveness conditions of a robust protocol.

Theorem 2.3.6: If there are no transmission errors, then the protocol of Figure 2.1 is optimal.

Proof: T and R each have two states. By Theorems 2.3.1 and 2.3.2, the number of states is

minimum. To show that the number of symbols in the transmission alphabet is the minimum, we assign values to the symbol names. One possible assignment — DATA = **0**, **1**; SYNC = ε; REQ = **0**; and ACK = **1** — yields a transmission alphabet consisting of the three symbols {**0**, **1**, ε}. By Theorem 2.3.5, three symbols are necessary. □

2.4. Transmission in the Presence of Deletion Errors

In this section, we consider data transmission in the presence of deletion errors alone. We present a robust protocol for deletion errors and show that it is optimal in the presence of read faults for both compulsive and selective machines. We then give a more efficient protocol for selective machines and write faults and establish its optimality. Finally we show that compulsive and selective finite-state machines are different.

2.4.1. Compulsive machines and read faults

For read faults, the compulsive stop-and-wait protocol of Figure 2.2 ensures reliable transmission in the presence of deletion errors. The protocol is basically the same as the protocol of Figure 2.1 except that if a machine reads an ε, then it does not change state. A machine reading an ε remains in its current state and resends the last symbol it wrote until it receives a nonnull symbol. The protocol is robust under deletion errors since the machines move through the same cycle of states as they do when there are no transmission errors. In general, any asynchronous protocol that does not send ε and is robust in the absence of transmission errors can be made robust for deletion errors by adding self-loops on ε at each state that resend the nonnull symbol the machine last sent.

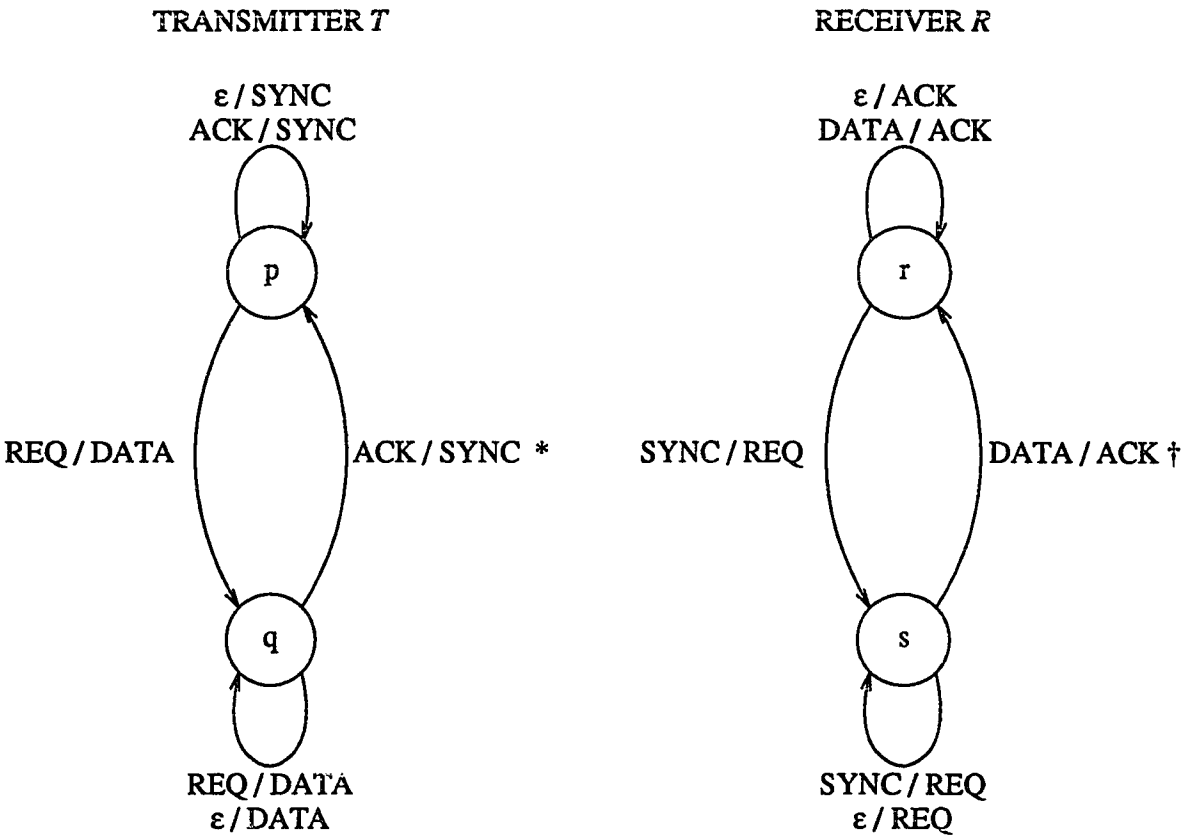

Figure 2.2. A robust protocol for deletion errors for compulsive machines.

In the remainder of this section, we establish a lower bound on the number of symbols required in the transmission alphabet in the presence of deletion errors and read faults. We then show that the protocol of Figure 2.2 is optimal.

Let T and R be the machines of a robust protocol for deletion errors. We may assume R is reduced. Fix $a \in \Sigma$, where a is nonnull. Let α be one of $\{a, \varepsilon\}$ and $\bar{\alpha}$ be the other. An $\{a, \varepsilon\}$-*sequence* of R is a sequence of distinct states $s_0, s_1, \ldots, s_k$ such that $\delta_R(s_i, \alpha) = s_{i+1}$ for

$0 \le i < j_1$, $\delta_R(s_{j_1}, \alpha) = s_{j_1}$, $\delta_R(s_i, \bar{\alpha}) = s_{i+1}$ for $j_1 \le i < j_2$, $\delta_R(s_{j_2}, \bar{\alpha}) = s_{j_2}$, $\delta_R(s_i, \alpha) = s_{i+1}$ for $j_2 \le i < j_3$, $\cdots$, $\delta_R(s_{k-1}, \alpha^*) = s_k$, where α^* is either α or $\bar{\alpha}$. That is, $s_0, \ldots, s_{j_1}$ is an α-sequence, $s_{j_1}, \ldots, s_{j_2}$ is an $\bar{\alpha}$-sequence, etc. An $\{a, \varepsilon\}$-*cycle* of R is a sequence of distinct states $s_0, s_1, \ldots, s_k$ such that states $s_0, s_1, \ldots, s_k$ form an $\{a, \varepsilon\}$-sequence and $\delta_R(s_k, \alpha^*) = s_0$. A state with self-loops on both a and ε is an $\{a, \varepsilon\}$-cycle with one state.

Lemma 2.4.1: R does not write into the output buffer in the transitions of an $\{a, \varepsilon\}$-cycle.

Proof: The proof is similar to the proof of Lemma 2.3.3. Suppose to the contrary that R does write into the output buffer in an $\{a, \varepsilon\}$-cycle. If R is in a state of an $\{a, \varepsilon\}$-cycle and a is in R's incoming cell, then since T and R are asynchronous and there are read faults, R may read an arbitrary sequence of a's and ε's before T makes its next move. Thus R may make an arbitrary number of transitions in the $\{a, \varepsilon\}$-cycle and output an arbitrary number of bits. □

For all a, a machine is $\{a, \varepsilon\}$-*reduced* if every $\{a, \varepsilon\}$-cycle has either one or two states.

Lemma 2.4.2: Suppose R is a reduced but not $\{a, \varepsilon\}$-reduced machine in a robust protocol for deletion errors. Then there is an $\{a, \varepsilon\}$-reduced machine R' such that if R is replaced with R', then the protocol with T and R' is robust.

Proof: Since the proof is similar to the proof of Lemma 2.3.4, we give a sketch. R' is obtained from R by breaking each $\{a, \varepsilon\}$-cycle with more than two states and forming an $\{a, \varepsilon\}$-cycle with two states.

Let $s_0, s_1, \ldots, s_k$ be an $\{a, \varepsilon\}$-cycle of R with more than two states. Let s_i be a state of the $\{a, \varepsilon\}$-cycle with a self-loop on a such that there is a transition from s_i on some nonnull symbol b that leads to a sequence of transitions in which R writes into the output buffer without first reentering a state of the $\{a, \varepsilon\}$-cycle, if any, or any state of the $\{a, \varepsilon\}$-cycle with

a self-loop on a otherwise; clearly $b \neq a$. Let s_j be a state of the $\{a, \varepsilon\}$-cycle with a self-loop on ε such that there is a transition from s_j on some nonnull symbol c that leads to a sequence of transitions in which R writes into the output buffer without first reentering a state of the $\{a, \varepsilon\}$-cycle, if any, and any state of the $\{a, \varepsilon\}$-cycle with a self-loop on ε otherwise; clearly $c \neq a$. Since R is robust, there is a sequence of transitions from at least one of s_i and s_j in which R outputs a bit. Let $b_i = \lambda_R(s_i, a)$ and $b_j = \lambda_R(s_j, \varepsilon)$.

To break the $\{a, \varepsilon\}$-cycle, replace the transition $\delta_R(s_i, \varepsilon)$ and its output by $\delta_{R'}(s_i, \varepsilon) = s_j$ and $\lambda_{R'}(s_i, \varepsilon) = b_j$, and replace the transition $\delta_R(s_j, a)$ and its output by $\delta_{R'}(s_j, a) = s_i$ and $\lambda_{R'}(s_j, a) = b_i$. By Lemma 2.4.1, R does not write into the output buffer during this transition. In this fashion, we break every $\{a, \varepsilon\}$-cycle and construct a machine R' whose only $\{a, \varepsilon\}$-cycles contain at most two states, for all a.

Next we show that if R is replaced with R', then the protocol with T and R' is robust. Suppose R' is in a state of the $\{a, \varepsilon\}$-cycle and a is in c_T. Since R' may read an arbitrary sequence of a's and ε's before T makes its next move, R' may leave the $\{a, \varepsilon\}$-cycle from state s_i or s_j. The execution of T and R' would be the same as an execution of T and R in which R leaves the $\{a, \varepsilon\}$-cycle from state s_i or s_j, respectively. □

By Lemma 2.4.2, we may assume without loss of generality that R is an $\{a, \varepsilon\}$-reduced machine for all nonnull $a \in \Sigma$.

Theorem 2.4.3: Any robust protocol for deletion errors for compulsive machines and read faults requires at least four symbols, including ε, in its transmission alphabet.

Proof: Suppose to the contrary that only two nonnull symbols **A** and **B** and ε suffice. Intuitively, sending an ε in the presence of deletion errors and read faults does not convey useful information because a machine receiving an ε cannot determine whether ε was sent or

a read fault occurred on a nonnull symbol. Thus machines should not send ε. The number of symbols in the transmission alphabet is then, in effect, reduced to the number of nonnull symbols.

More precisely, consider an execution of R in which whenever T writes a nonnull symbol a, R makes transitions on a and ε caused by read faults until R enters a state of an $\{a, \varepsilon\}$-cycle. By Lemma 2.4.1, R must leave the cycle to output another bit.

Let α be one of {A,B} and β be the other. Suppose R reads a sequence of α's and ε's and enters a state of an $\{\alpha, \varepsilon\}$-cycle. Since there are only three symbols, R can leave the cycle only on β. If the $\{\alpha, \varepsilon\}$-cycle has one state, then there is only one way R can leave the cycle. If the $\{\alpha, \varepsilon\}$-cycle has two states, then R can leave the cycle in two ways. We consider two cases.

Case 1: While R is in the $\{\alpha, \varepsilon\}$-cycle, if T sends β only after T sends and R receives ε, then the only way R can leave the $\{\alpha, \varepsilon\}$-cycle is on a transition on β from the state of the cycle with a self-loop on ε.

Case 2: While R is in the $\{\alpha, \varepsilon\}$-cycle, if T can send β immediately after sending α, then R can leave the cycle either from the state with a self-loop on α, or if a read fault occurs, from the state with a self-loop on ε. The latter happens if R reads its incoming cell twice before T moves and a read fault occurs on the first read. If R reads an ε, then R cannot determine whether T wrote ε or T wrote β and a deletion error occurred. Thus, if R leaves the $\{\alpha, \varepsilon\}$-cycle on a transition on β from the state with a self-loop on ε, then the next bit R outputs must be the same as the one R would have output if R had left the cycle from the state with a self-loop on α. A fortiori, all subsequent bits that are output must be the same.

The sequence of bits output by R is fixed after the first. By the proof of Theorem 2.3.5, the output of R has the prefix property. Thus, there is no robust protocol for deletion errors for compulsive machines and read faults that uses only two nonnull symbols and ε in its transmission alphabet. □

We now establish the optimality of the protocol shown in Figure 2.2.

Theorem 2.4.4: The protocol of Figure 2.2 is optimal for compulsive machines and read faults.

Proof: By Theorems 2.3.1 and 2.3.2, the protocol has the minimum number of states. To show that the number of symbols in the transmission alphabet is the minimum, we assign values to the symbol names. One possible assignment — DATA = **0**,**1**; SYNC = **2**; REQ = **0**; ACK = **1**; and $\varepsilon = \varepsilon$ — yields a transmission alphabet consisting of the four symbols {**0**, **1**, **2**, ε}. By Theorem 2.4.3, four symbols are necessary. □

2.4.2. Selective machines and read faults

The protocol of Figure 2.2 is also robust for selective machines and read faults since a compulsive machine is simply a selective machine that writes on every transition. The protocol is also optimal for selective machines since the sequence of symbols read by T and R in the presence of read faults when they are selective machines can be the same as when they are compulsive machines.

The protocol for selective machines could be slightly simplified, however, since a selective machine does not have to send a symbol on every transition. Since write operations are reliable, a machine needs to send a symbol only once. Thus, we could modify the

protocol of Figure 2.2 and eliminate writes on all self-loops, except for the initial ε / SYNC transition for T to get the protocol started.

2.4.3. Selective machines and write faults

The protocol of Figure 2.3 is robust for deletion errors for selective machines and write faults. A dash (—) indicates that a machine does not write a symbol on the transition. The

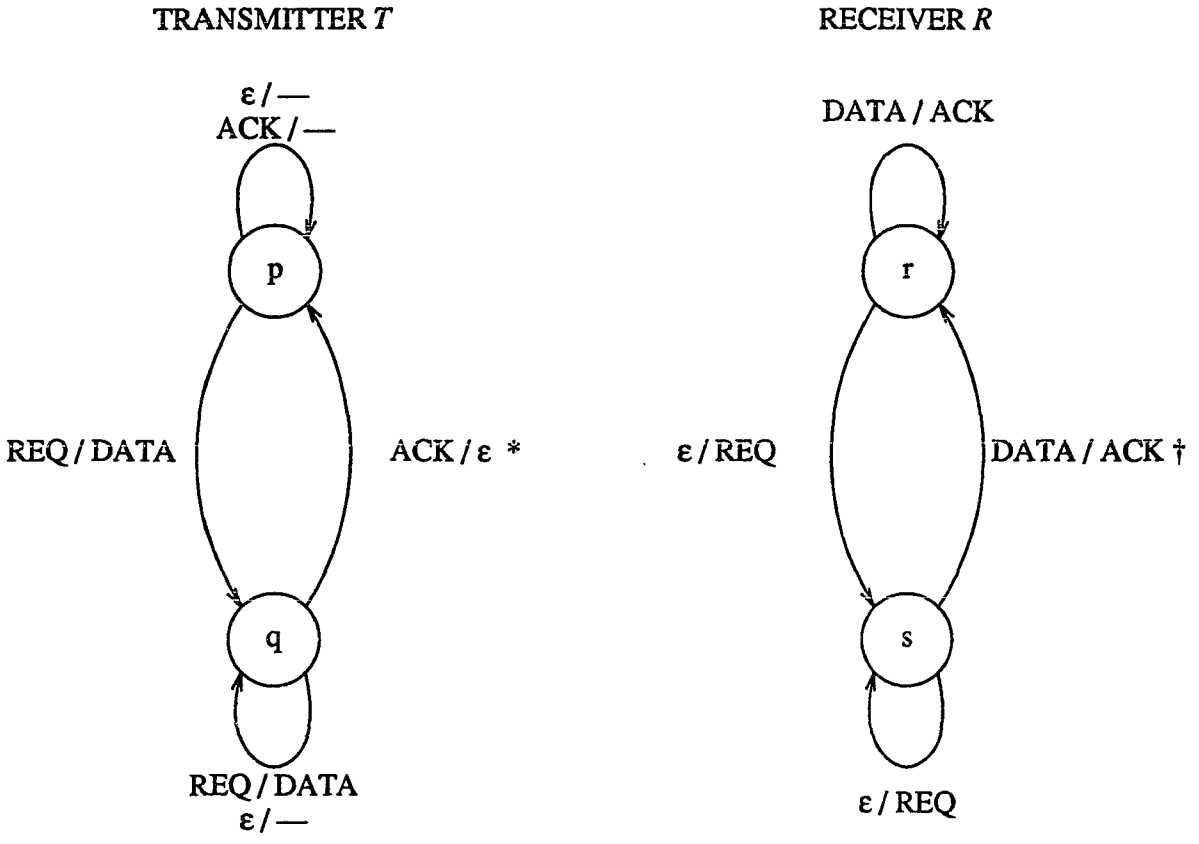

Figure 2.3. A robust protocol for deletion errors for selective machines and write faults.

protocol of Figure 2.3 is essentially the protocol of Figure 2.2 with the SYNC symbol replaced by ε. The main difference is that in the protocol of Figure 2.3 machines can be made to wait. Since R sends only nonnull symbols, if T receives an ε, then T remains in its current state until R successfully resends the last symbol.

Theorem 2.4.5: The protocol of Figure 2.3 is optimal for selective machines and write faults.

Proof: By Theorems 2.3.1 and 2.3.2, the protocol has the minimum number of states. To show that the number of symbols in the transmission alphabet is the minimum, we assign values to the symbol names. One possible assignment — DATA = **0**,**1**; REQ = **0**; ACK = **1**; and ε = ε — yields a transmission alphabet consisting of the three symbols {**0**, **1**, ε }. By Theorem 2.3.5, three symbols are necessary. □

2.4.4. Compulsive machines and write faults

It is straightforward to verify that the protocol of Figure 2.2 is robust under deletion errors for compulsive machines and write faults. We showed that for selective machines and write faults there is a robust protocol for deletion errors in which T and R each have two states and the transmission alphabet consists of three symbols. In contrast, we show in this section that for compulsive machines and write faults there is no robust protocol for deletion errors in which the transmitter and receiver each have two states and the transmission alphabet has three symbols. Thus, the compulsive and selective models are different.

Theorem 2.4.6: For compulsive machines and write faults there is no robust protocol for deletion errors that consists of a two-state transmitter, a two-state receiver, and a transmission alphabet with three symbols.

Proof: Suppose to the contrary that there is such a protocol. We first show that any robust two-state receiver must be similar to the receiver in the protocol of Figure 2.3.

Let R be the receiver and let the two states of R be r and s. By Lemma 2.3.3, any buffer operation performed by R must occur on a transition between the two states. Without loss of generality, assume that if R is in state s and reads a, then R makes a transition to state r, writes **0** into its output buffer, and sends x to T. Then R has no transition from state r to s on a; otherwise, R could output an arbitrary number of **0**'s. Thus, state r has a self-loop on a.

Next, consider a transition of R that writes a **1** into the output buffer. It can be seen that there must be a transition either from state r to s or from state s to r that occurs on some symbol b, where $b \neq a$. We show that R cannot have a transition from r to s on b that writes **1** into the output buffer.

Suppose to the contrary that R does have such a transition. Then state s has a self-loop on b. If R is in state r and must next output a **0**, then R must make a transition to state s on some third symbol c, where $c \neq a$ and $c \neq b$. Similarly, if R is in state s and must next output a **1**, then R must make a transition to state r on some symbol d, where $d \neq a$ and $d \neq b$. Since there are only three symbols, $d = c$. Thus R has a c-cycle.

By Lemma 2.3.3, R does not write into the output buffer on the transitions of the c-cycle. Suppose R is in state r and must next output a **0**. Then R must make an odd number of transitions on c before T makes its next move and sends an a. But since T and R are asynchronous, R may make an even number of moves and fail to output a **0**. The next bit that R outputs may then violate the safety condition.

Thus, the buffer operation for R to write a **1** into the output buffer occurs on a transition from state s to r on b. Let y be the symbol that R sends to T on the transition. Note that y may be the same as x. To avoid a b-cycle, state r has a self-loop on b.

Finally, R needs a transition from state r to return to state s. The transition occurs on some symbol c, where $c \neq a$ and $c \neq b$. Let z be the symbol that R sends to T on the transition. To avoid a c-cycle that may cause R to fail to make an output, state s has a self-loop on c.

Now we assign values to the symbols a, b, and c. Since there are deletion errors, a and b must be nonnull. Otherwise, if R is in state s and a deletion error occurs, then R may output the wrong bit. Thus c must be ε. We have now established that any robust two-state receiver is similar to the receiver in the protocol of Figure 2.3.

Next, we consider the structure of the transmitter. Let T be the transmitter and let the two states of T be p and q. Transmitter T communicates the current input bit to R by sending a symbol to R. By the structure of R, the symbol must be nonnull. Since there are deletion errors, however, if the symbol that T sends to R is deleted, then T must be able to resend that symbol. Thus T cannot advance its input pointer on the transition in which T first sends a symbol to communicate the current input bit to R. Otherwise, T may skip an input bit.

From the above restriction, the structure of R, and the requirement that T advance its input pointer on a transition between two different states, we infer that T communicates the current input bit by sending either a or b, and T advances its input pointer on a transition between states p and q on either x or y.

Without loss of generality, assume that T advances its input pointer on the transition from state q to state p on x. Given the structure of R, T sends c on the transition. To avoid an x-cycle that may cause T to skip input bits, state p has a self-loop on x. Similarly, there are corresponding transitions for y. To allow T to return to state q, there must be a transition from state p to q on some other symbol z. Given the structure of R, T first sends a symbol to communicate the current input bit to R on a transition from state p to q. Thus there are two transitions, one for communicating a **0** and one for communicating a **1**. To avoid a z-cycle that can cause T to communicate an input bit more than once and to allow T to resend a symbol in the case of a deletion error, state q has two self-loops on z, one for communicating a **0** and one for communicating a **1**.

We now show that the construction of T cannot be completed while preserving the robustness of the protocol. Since there are deletion errors, T must have transitions on ε from each of states p and q. We show that there is no transition from state q on ε that preserves the robustness of the protocol. We consider three cases.

Case 1: T has a transition from q to p on ε and T does not advance its input pointer on the transition. If a deletion error occurs when R sends x or y, then T does not advance its input pointer after communicating the current input bit, and eventually R will output an extra bit.

Case 2: T has a transition from q to p on ε and T advances its input pointer on the transition. Then T may advance its input pointer before communicating the current input bit to R as follows: T sends a to R and the a is deleted, and then the z that R sends to T is deleted.

Case 3: T has a self-loop on ε in state q. Consider the following execution. Let the current input bit be **0**. Suppose T is in state q and sends a to R. When R receives a, R moves to state r, writes **0** into the output buffer, and sends x to T. Now a deletion error occurs and T receives ε. Since T is compulsive, T must send some symbol to R, but another deletion error occurs. When R receives ε, R moves to state s and sends z to T. When T receives z, T sends a to R. When R receives a, R writes an extra **0** into the output buffer. □

Note that in the selective model the self-loop on ε in state q does not cause extra bits to be output because when T receives ε, T does not have to send another symbol to R. Thus the nonnull symbol T sent to R remains in R's incoming cell.

We have shown that there is no robust protocol for deletion errors for compulsive machines and write faults that consists of a two-state transmitter and a two-state receiver that uses only three symbols. The protocol of Figure 2.2 is a protocol that uses four symbols. We conjecture that any robust protocol for deletion errors for compulsive machines and write faults requires four symbols, regardless of the number of states.

We have thus constructed robust protocols for deletion errors for both compulsive and selective machines, and for both read and write fault models of the shared memory. We have established the optimality of the protocols for read faults and for selective machines and write faults. We have also shown that compulsive and selective machines are different.

2.5. Transmission in the Presence of Insertion Errors

We first consider transmission in the presence of insertion errors and read faults.

Theorem 2.5.1: Any robust protocol for insertion errors and read faults requires at least three nonnull symbols in its transmission alphabet.

Proof: Suppose to the contrary that only ε and two nonnull symbols **A** and **B** are necessary. If T sends ε, then R may receive ε or any other nonnull symbol. Since T and R are asynchronous, if T writes ε, then R may read an arbitrary sequence of ε's, **A**'s, and **B**'s caused by insertion errors before T makes its next move. Thus R may read a sequence of symbols that causes R to output the wrong bit. If T never sends ε, then the transmission alphabet is reduced to two symbols. By Theorem 2.3.5, three symbols are necessary. □

Theorem 2.5.2: The protocol of Figure 2.1 is robust for insertion errors and read faults and optimal.

Proof: The protocol of Figure 2.1 is robust for insertion errors since neither machine sends ε. To show that the number of symbols in the transmission alphabet is the minimum, we assign values to the symbol names. One possible assignment — DATA = **0**, **1**; SYNC = **2**; REQ = **0**; and ACK = **1** — yields a transmission alphabet consisting of the three nonnull symbols {**0**,**1**,**2**} and ε. It follows from Theorems 2.3.1, 2.3.2, and 2.5.1 that the protocol is optimal. □

For compulsive machines and write faults, the protocol of Figure 2.4 is robust under insertion errors and uses only two nonnull symbols and ε. The pound sign (#) represents any nonnull symbol. The protocol of Figure 2.4 is essentially the protocol of Figure 2.1 with SYNC replaced by ε. When R is in state r, R expects to receive ε. If an insertion error occurs and R receives a nonnull symbol, then R remains in state r until it receives ε. It follows from Theorems 2.3.1, 2.3.2, and 2.3.5 that the protocol is optimal.

For selective machines and write faults, there is a robust protocol for deletion and insertion errors that uses two nonnull symbols and ε. We present the protocol in the next section.

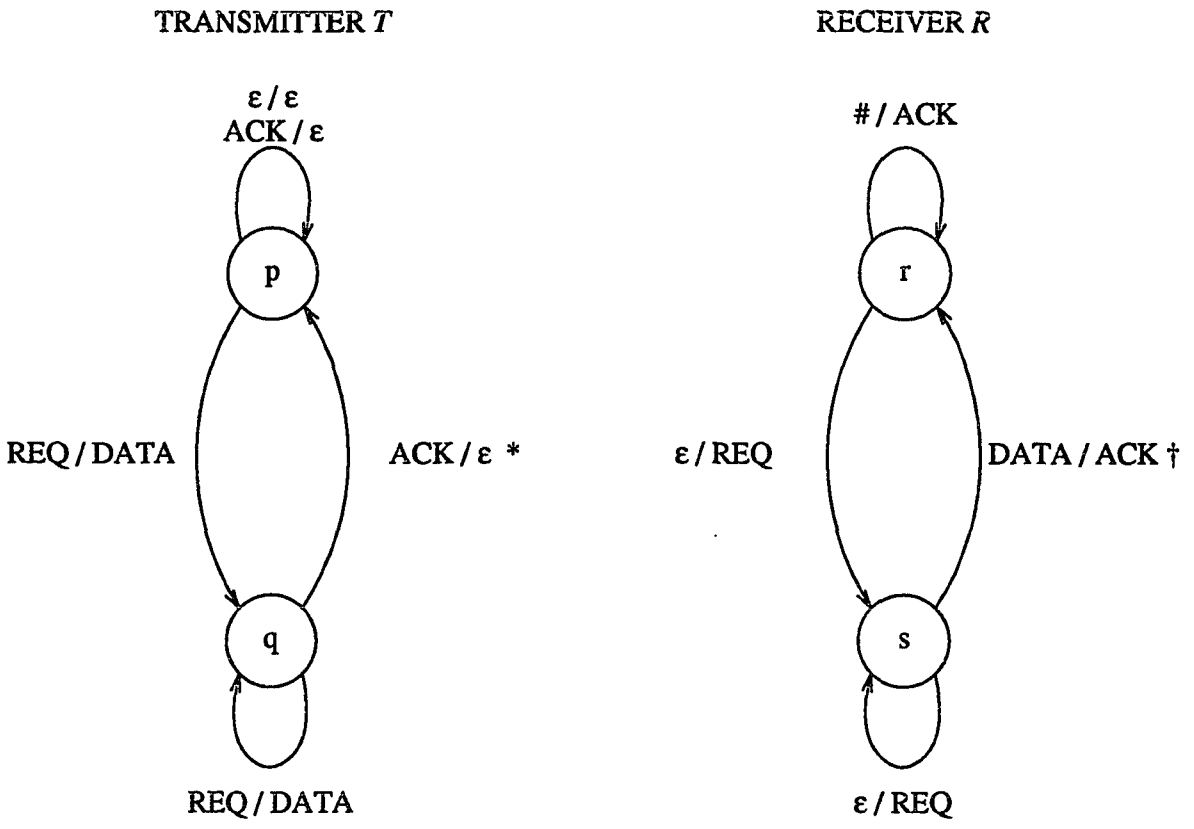

Figure 2.4. A robust protocol for insertion errors for compulsive machines and write faults.

2.6. Transmission in the Presence of Mutation and Multiple Errors

We first consider transmission in the presence of mutation errors and then in the presence of deletion and insertion errors.

Theorem 2.6.1: There is no robust protocol in the presence of mutation errors.

Proof: Consider the symbols sent by T and R. Since there are mutation errors, a nonnull symbol sent by T may be received as any nonnull symbol by R. Thus the transmission

alphabet is, in effect, reduced to only two symbols, ε and #, where # represents any nonnull symbol of the transmission alphabet. By Theorem 2.3.5, at least three symbols are necessary. Thus, there is no robust protocol for mutation errors. □

Since there are no robust protocols for mutation errors, there are also no robust protocols for combinations of errors that include mutation errors. The only combination remaining is deletion and insertion errors. But it can be seen that the protocol of Figure 2.1 for compulsive machines is also robust in the presence of both deletion and insertion errors. Since T and R do not send ε, the deletion and insertion errors case is, in effect, reduced to the deletion errors only case. Thus the protocol is robust even if both deletion and insertion errors are present. Clearly, the protocol is also robust for selective machines.

For the case of selective machines and write faults, however, the protocol of Figure 2.5 requires only three symbols, including ε, in its transmission alphabet. The protocol of Figure 2.5 is very similar to the protocol of Figure 2.4. Since T expects only nonnull symbols, if T receives an ε, then T remains in its current state and waits for a nonnull symbol. By Theorems 2.3.1, 2.3.2, and 2.3.5, the protocol is optimal.

2.7. Conclusions

We have modeled asynchronous communication protocols with communicating finite-state machines and established lower bounds on the size of machines and the number of symbols needed in the transmission alphabet to ensure reliable communication. We carefully studied two different types of machines, compulsive and selective, and two fault models for the shared memory, the read fault model and the write fault model.

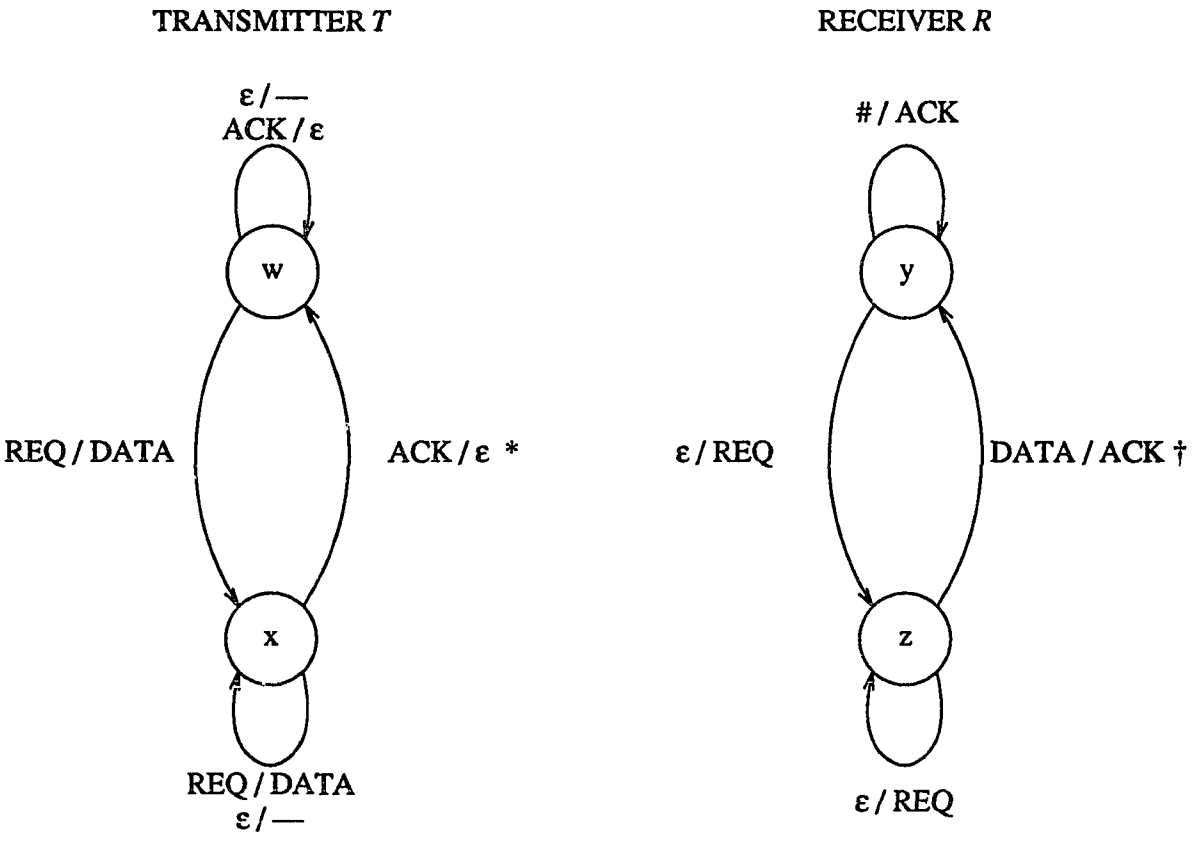

Figure 2.5. A robust protocol for deletion and insertion errors for selective machines and write faults.

We summarize our results and the results of Aho et al. (1982) for synchronous protocols in Table 2.1. An entry containing a triple x, y, z indicates that there is a robust protocol for that class of errors and that the protocol has x states for the transmitter, y states for the receiver, and z symbols in the transmission alphabet. An asterisk (*) indicates the asynchronous protocols that are optimal in both the size of the machines and the number of symbols in the transmission alphabet. A "Yes" entry appears for the results of Aho et al.

Table 2.1. *Existence of robust protocols in the presence of various errors.*

Errors	Synchronous	Asynchronous			
		Compulsive Machines		Selective Machines	
		Read Fault	Write Fault	Read Fault	Write Fault
Deletion	2, 2, 3	2, 2, 4*	2, 2, 4	2, 2, 4*	2, 2, 3*
Insertion	Yes	2, 2, 4*	2, 2, 3*	2, 2, 4*	2, 2, 3*
Mutation	Yes	No	No	No	No
Del.-Ins.	Yes	2, 2, 4*	2, 2, 4	2, 2, 4*	2, 2, 3*
Del.-Mut.	3, 3, 2	No	No	No	No
Ins.-Mut.	Yes	No	No	No	No
Del.-Ins.-Mut.	No	No	No	No	No

where they established only the existence of robust protocols for a class of errors. A "No" entry appears for those classes of errors where no robust protocol exists.

We have shown that for each combination of machine type and memory fault there is a robust asynchronous communication protocol for deletion and insertion errors. We established that the protocols for compulsive machines in the read fault model and for selective machines in both the read and write fault models have the minimum number of states for the machine and the minimum number of symbols in the transmission alphabet simultaneously. For compulsive machines in the write fault model, our protocol uses the minimum number of symbols when the transmitter and receiver have only two states each. We also showed that there are no robust asynchronous protocols for mutation errors. In contrast, robust synchronous protocols for mutation errors exist. Finally, our results illustrate

the differences between read faults and write faults, and between compulsive machines and selective machines.

One area of further research is to investigate protocols for restricted types of mutation errors. For example, mutation errors may be limited such that a symbol that is sent is either received correctly or mutated to one of a subset of symbols in the transmission alphabet. Other restrictions can be placed to limit the number of different symbols that may be mutated to the same symbol.

CHAPTER 3.

PARALLEL ALGORITHMS FOR MINIMUM SPANNING TREE IN LOGARITHMIC TIME WITHOUT PRIORITY WRITING

3.1. Introduction

Let $G = (V, E)$ be a connected graph with a set of vertices V and a set of edges E in which each edge e has a weight $w(e)$. Without loss of generality, assume that the edge weights are distinct. Hence, the minimum spanning tree (MST) of G is unique. Let $n = |V|$ and $m = |E|$. We present an algorithm for finding the MST of G on a Common Concurrent Read Concurrent Write (CRCW) Parallel Random Access Machine (PRAM) in $O(\log n)$ time using $2m + n^{1+2\varepsilon}$ processors, where ε is a constant such that $0 < \varepsilon \leq 1/2$.

The MST problem is an important problem of combinatorial optimization. Some practical applications of MSTs include the design of computer, communication, and transportation networks. Graham and Hell (1985) gave an extensive history of the MST problem.

Yao (1975) and Cheriton and Tarjan (1976) designed sequential MST algorithms that run in time $O(m \log\log n)$. Fredman and Tarjan (1987) gave an improved algorithm which runs in time $O(m\beta(m,n))$, where $\beta(m,n) = \min \{i \mid \log^{(i)} n \leq \frac{m}{n}\}$. If $m \geq n$, then $\beta(m,n) \leq \log^* n$. Gallager et al. (1983) presented a distributed MST algorithm that uses at most $5n\log n + 2m$ messages and $O(n \log n)$ time. Awerbuch (1987) presented an optimal distributed MST algorithm that uses $O(m + n \log n)$ messages and $O(n)$ time. The algorithm is optimal in both communication and time.

There have also been several parallel MST algorithms. Chin et al. (1982) presented an efficient Concurrent Read Exclusive Write (CREW) PRAM algorithm that runs in $O(\log^2 n)$ time using $(\frac{n^2}{\log^2 n})$ processors. Hirschberg (1982) gave an algorithm for the Common CRCW PRAM which runs in $O(\log n)$ time using n^3 processors. Awerbuch and Shiloach (1987) designed an algorithm for the Priority CRCW PRAM which runs in $O(\log n)$ time using $O(m+n)$ processors. Cole and Vishkin (1986a) presented a Priority CRCW PRAM algorithm that runs in $O(\log n \log^{(2)} n \log^{(3)} n)$ time using $\frac{(m+n)}{\log n \log^{(2)} n}$ processors. Cole and Vishkin (1986b) then gave a slightly faster Priority CRCW PRAM algorithm that runs in $O(\log n)$ time using $\frac{(m+n)\log^{(3)} n}{\log n}$ processors.

Our MST algorithm is based on modifications of the algorithm of Awerbuch and Shiloach. We employ some of the results of Fich et al. (1988a) to obtain a Common CRCW PRAM MST algorithm. A straightforward modification yields an algorithm that runs in $O(\log n)$ time using $mn + n^{1+2\varepsilon}$ processors. We then reduce the number of processors to $2m + n^{1+2\varepsilon}$. Our algorithm has the same running time as the algorithm of Hirschberg but uses fewer processors. For mildly dense graphs, where $m = \Omega(n^{1+2\varepsilon})$, our algorithm has the same performance as the algorithm of Awerbuch and Shiloach using a weaker CRCW PRAM model.

In Section 3.2, we describe the model of computation. In Section 3.3, we review the MST algorithm of Awerbuch and Shiloach. In Section 3.4, we present our Common CRCW PRAM algorithms.

3.2. Model of Computation

A PRAM consists of a set of reliable synchronous processors, each with its own local memory, and a global shared memory. Each step of a processor of the PRAM consists of three *stages*. In the first stage, each processor may read a value from one shared memory cell into local memory. In the second stage, each processor may perform a local computation on values in its local memory. In the third stage, each processor may write a value from local memory into one shared memory cell. In the CRCW PRAM, any number of processors may simultaneously read from the same shared memory cell during a read stage, and any number of processors may simultaneously write to the same shared memory cell during a write stage. If more than one processor writes into the same cell, then the value that is written depends on the model.

Two CRCW PRAM models are the Priority model and the Common model. For each model, we describe what happens when two or more processors simultaneously try to write into the same cell. In the Priority model, each processor has a unique fixed priority. The processor with the highest priority is the one that succeeds. In the Common model, all processors must write the same value. Fich et al. (1988a), Fich et al. (1988b), Chlebus et al. (1988), and Boppana (1989), among others, have studied the relationships between the two CRCW PRAM models.

3.3. The MST Algorithm of Awerbuch and Shiloach

The algorithm of Awerbuch and Shiloach (1987) uses a Priority CRCW PRAM. The priority of each processor is determined by its index. The smaller the index, the higher its priority.

Each edge of G is represented by two oppositely directed edges. The algorithm assigns processors to edges such that the smaller the weight of an edge, the higher the priority of the corresponding processor. The assignment can be made by sorting the edges by weight and then assigning processors in order. Let $p(i,j)$ and $p(j,i)$ be the processors assigned to edge (i,j). This can be done in $O(\log n)$ time using the parallel merge sort algorithm of Cole (1988).

Without loss of generality, assume that the vertices of G are numbered from 1 to n. The number of a vertex is its identifier, denoted *id*. We will refer to vertices by their *id*'s. The notation $u < v$ means that the *id* of vertex u is smaller than the *id* of vertex v.

In the algorithm, there are variables associated with each vertex i. The processors that operate on these variables, however, correspond to edges. Each vertex i has a *parent* $P(i)$, which is either another vertex or itself. If a vertex is a root, then its parent is itself. The parent-child relation defines a directed graph called the *parents graph*, PG. PG has the same vertices as G. Define $GP(i) = P(P(i))$, and call $GP(i)$ the *grandparent* of i.

The algorithm maintains a set T of undirected edges which always forms a forest of the MST. The algorithm adds edges to T using the property that for any subset of vertices, the edge of least weight leaving the set must belong to the MST. Set T grows until it becomes the MST.

A *rooted tree* is a tree whose edges are directed toward the *root*. A *star* is a rooted tree with height 1. The algorithm maintains the invariant that after each iteration, for each directed tree in PG, there is a subtree in T spanning the same set of vertices. The algorithm finds edges of the MST by trying to combine trees in PG. Two trees are combined by *hooking* one tree to the other. The algorithm hooks a tree T_1 to a tree T_2 by making some

vertex of T_2 the parent of the root of T_1. Processors that correspond to edges leaving a star try to hook the star to a tree. Edges that correspond to successful processors are added to T. After the stars are hooked, the algorithm reduces the height of each tree with a shortcut operation, where each vertex takes its grandparent to be its new parent.

A Boolean variable $T(e)$ is attached to each edge e; $T(e)$ is initially 0. During the algorithm, if edge e is added to the T, then $T(e)$ is set to 1. *Winner*(i) contains the name of the edge corresponding to the writing processor. After the initialization, the algorithm iterates three Parts until all of the vertices are in the same star in PG. The algorithm is executed in parallel by each edge processor $p(i,j)$.

Priority CRCW PRAM Minimum Spanning Tree Algorithm

Initialization

$T(e) := 0$ for all $e \in E$;
$P(i) := i$ for $i = 1, \ldots, n$;

For each processor $p(i,j)$

repeat

Part 1: (Star hooking)
if i belongs to a star **and** $P(i) \neq P(j)$ **then**
$P(P(i)) := P(j)$;
Winner$(P(i)) := (i,j)$;
endif
if *Winner*$(P(i)) = (i,j)$ **then**
$T(i,j) := 1$;
endif

Part 2: (Cycle breaking)
if $i < P(i)$ **and** $i = GP(i)$ **then**
$P(i) := i$;
endif

Part 3: (Shortcut operation)
$P(i) := GP(i)$;

until every vertex i belongs to the same star

Part 1 performs the hooking operation. Processors that correspond to edges leaving a star try to hook the star to another tree. A star is hooked to a tree by setting the parent of the root of the star to a vertex of the tree to which the star is being hooked. If more than one processor tries to hook the star, then the processor with the highest priority succeeds. Thus, $Winner(i)$ contains the name of the edge e of smallest weight leaving the star. Since edge e belongs to the MST, the algorithm sets $T(e) := 1$. At the end of Part 1, every star is hooked to some tree.

Part 2 eliminates any cycles that may have been formed in the parent's graph. A cycle of length two forms when an edge's endpoints belong to two different stars and the edge is the edge of least weight leaving both stars. To break a cycle, the algorithm changes the parent pointer of the vertex with the smaller *id* to point to itself.

Part 3 performs the shortcut operation. For each vertex i, the algorithm sets the grandparent of i to be the new parent of i. Note that if more than one processor updates $P(i)$, then all processors write the same value. The height of each tree that is not a star decreases by a factor of at least 3/2.

A vertex determines whether it belongs to a star by using Procedure Star_Check. At the termination of Star_Check, if $ST(i)$ is **true** (**false**), then i belongs (does not belong) to a star.

Procedure Star_Check

For each processor $p(i)$

$ST(i) :=$ **true;**
if $P(i) \neq GP(i)$ **then**
    $ST(i) :=$ **false;**
    $ST(GP(i)) :=$ **false;**
endif
$ST(i) := ST(P(i))$;

Awerbuch and Shiloach established the correctness of their algorithm. We briefly verify the running time. Consider each iteration of the three Parts. Parts 1 and 2 ensure that every star is hooked to some tree to yield a new tree with height greater than one. Since Part 3 reduces the height of every tree with height greater than one by a factor of at least 3/2, the sum of the heights of all of the trees present at the start of the iteration is reduced by a factor of at least 3/2. Thus, O(log n) iterations yield a single star. Since each iteration takes O(1) time, the algorithm runs in O(log n) time.

3.4. Common CRCW PRAM MST Algorithm

Our Common CRCW PRAM MST algorithm is the same as the algorithm of Awerbuch and Shiloach except that Part 1 is modified to eliminate the priority concurrent write. Thus we describe the modified implementation of Part 1 only. In our algorithm, we avoid the priority write by determining the processor of highest priority wanting to write to each memory cell and having only those processors write. It can be seen that the values written in the memory cells are the same as those that would have been written in the Priority CRCW PRAM model.

To determine the processor of highest priority writing to each cell, we solve a special case of the r-color minimization problem described in Fich et al. (1988a).

r-Color Minimization Problem

Before: Each processor p_i, $i = 1, \cdots, p$, has a color x_i, $0 \le x_i \le r$, known only to itself. Color x_i represents the cell p_i wants to write, if any, and 0 otherwise.

After: Each processor p_i knows the value a_i, where $a_i = 1$ if and only if p_i is the processor of lowest index writing into the cell represented by x_i.

For our algorithm, we consider the case in which $r = 1$. Fich et al. showed that on a Common CRCW PRAM with k memory cells, the 1-color minimization problem can be solved in $O(\frac{\log p}{\log (k+1)})$ steps. In our discussion, we present a simplified variation of their method and show how the problem can be solved in $O(\frac{\log p}{\log k})$ steps.

Let $M_1, \cdots, M_k$ be the k memory cells. Assume without loss of generality that $k \le p^{\varepsilon}$, where ε is a constant such that $0 < \varepsilon \le 1/2$. If $k > p^{1/2}$, then only the first $p^{1/2}$ cells are needed to achieve O(1) steps. The algorithm iterates the following steps. Processor p_i, $i = 1, \ldots, k$, writes 0 into M_i. The processors are then divided into k groups of nearly equal size, where each group is a set of consecutively numbered processors. The first $p \mod k$ groups contain $\left\lceil \frac{p}{k} \right\rceil$ processors, and the remaining groups contain $\left\lfloor \frac{p}{k} \right\rfloor$ processors. A processor p_i in the jth group, $1 \le j \le k$, writes 1 into M_j if and only if $x_i = 1$.

The *winner* is the processor of smallest index with $x_i = 1$. Thus the winner is in the group corresponding to the M_j of smallest index containing 1. The algorithm determines the winning group by using the subroutine Leftmost One.

Leftmost One

Before: Cells $M_i, i = 1, \ldots, k$, each contain 0 or 1.

After: M_i contains 1 if and only if all M_j for $j < i$ were initially 0, and M_i was initially 1.

The Leftmost One algorithm compares all pairs of cells M_i and M_j, $1 \leq i, j \leq k$. If $j < i$ and M_i and M_j both contain 1, then the algorithm writes 0 into M_i. The algorithm requires $k^2 \leq p$ processors. After applying the Leftmost One subroutine, processors in group j read M_j. A group determines it is the winning group if its processors read a 1.

All processors that are not in the winning group set $a_i := 0$ and stop. The processors in the winning group then repeat the 1-color minimization algorithm. This process repeats until the winning group contains only one processor, the winner.

Each iteration of the 1-color minimization algorithm reduces the number of processors that may be the winner by a factor of k. Thus the winner is determined in at most $\lceil \log_k p \rceil$ iterations. Since each iteration takes O(1) steps, the winner is determined in $O(\frac{\log p}{\log k})$ steps.

In Part 1 of the Priority CRCW PRAM algorithm, if more than one processor tries to hook a star with root i to a tree, then a priority write of the variable $P(i)$ occurs. Since there is a $P(i)$ for each vertex i, there are n cells into which processors may write. Since processors performing the hooking operation correspond to edges leaving stars, as many as m processors may want to write into one $P(i)$.

In the Common CRCW PRAM algorithm, we first determine the processor of highest priority writing to each $P(i)$ and then have only that processor write. We begin with the direct implementation which requires solving the r-color minimization problem with m processors and n colors.

To maintain the O(log n) running time of the MST algorithm, Part 1 must run in time O(1). In Part 1, the Common PRAM algorithm simultaneously solves n 1-color minimization problems, one for each $P(i)$. Each problem requires $m + n^{2\varepsilon}$ processors to obtain an O(1) time solution. During the first iteration, m processors are divided into n^{ε} groups. During each iteration, a processor can determine the group to which it belongs since it knows its rank from the sort performed during the initialization phase. Each iteration reduces the number of contending processors by a factor of n^{ε}, and thus $O(\frac{\log m}{\log n^{\varepsilon}}) = O(1)$ iterations suffice. Since there are n problems, Part 1 requires a total of $mn + n^{1+2\varepsilon}$ processors. We now show how to reduce the number of processors.

In Part 1, each processor corresponding to an edge leaving a star writes to exactly one $P(i)$. Thus each processor that wants to write is a possible winner for only one of the n 1-color minimization problems.

The absence of nonwriting processors from the groups of processors formed during the solution of the 1-color minimization problem does not affect the outcome since the processors would not have written even if they were present. Thus each processor that wants to write needs only to participate in the solution of the 1-color minimization problem corresponding to the $P(i)$ it wants to write. There are $2m$ edge processors that may write. Hence, for the n 1-color minimization problems, the algorithm requires a total of only $2m+n^{1+2\varepsilon}$ processors.

The remaining Parts of the algorithm require $2m + n$ processors and only common write operations. Thus we have a Common CRCW PRAM algorithm for the MST problem that runs in O(log n) time using $2m + n^{1+2\varepsilon}$ processors, where $0 < \varepsilon \leq 1/2$.

CHAPTER 4.

ASYNCHRONOUS PARALLEL ALGORITHMS FOR GRAPH CONNECTIVITY AND RELATED PROBLEMS

4.1. Introduction

Let $G = (V, E)$ be an undirected graph with the set of vertices V and the set of edges E. Let $n = |V|$ and $m = |E|$. A *subgraph* of G is a graph whose vertices and edges are in G. A *connected component* of G is a maximal connected subgraph of G.

Finding the connected components of a graph is a fundamental problem of graph theory. Some practical applications of graph connectivity algorithms include the detection of failures in computer networks, communication networks, and distributed database systems. On single processor machines, sequential algorithms can perform depth-first search to find the connected components of G in $O(m+n)$ time. To find the connected components much faster, however, requires parallel computers, where multiple processors can work together on the problem concurrently. One well-studied model of parallel computation is the Parallel Random Access Machine (PRAM).

For the Concurrent Read Exclusive Write (CREW) PRAM, Hirschberg et al. (1979) presented a connected components algorithm that runs in $O(\log^2 n)$ time using $O(\frac{n^2}{\log n})$ processors. Chin et al. (1982) improved their result by reducing the number of processors to $O(\frac{n^2}{\log^2 n})$. Han and Wagner (1990) presented a more efficient and nearly optimal algorithm that runs in time $O(\frac{m}{p} + \frac{(n \log n)}{p} + \log^2 n)$, where p is the number of processors. Their

algorithm achieves linear speedup when $p \leq \frac{m}{\log^2 n}$ and $m \geq n \log n$. Johnson and Metaxas (1991) designed the fastest known CREW PRAM algorithm. Their algorithm runs in $O(\log^{3/2} n)$ time using $m + n$ processors.

For the weaker Exclusive Read Exclusive Write (EREW) PRAM, Kruskal et al. (1990) gave an algorithm that computes the connected components of a graph in $O(\frac{m}{p} \frac{\log n}{\log(m/(p^2 \log p))} + \frac{n}{p} \log p)$ time using p processors. Their algorithm achieves linear speedup for $m = \Omega(n \log p)$ and $m = \Omega(p^{2+\varepsilon})$ for any constant $\varepsilon > 0$.

For the Arbitrary Concurrent Read Concurrent Write (CRCW) PRAM, Shiloach and Vishkin (1982) presented an algorithm that runs in $O(\log n)$ time using $O(m + n)$ processors. Awerbuch and Shiloach (1987) gave a similar and slightly simpler algorithm with the same performance. The latter algorithm also has a substantially simpler correctness proof. Cole and Vishkin (1986a) presented an Arbitrary CRCW PRAM algorithm that runs in $O(\log n \log^{(2)} n \log^{(3)} n)$ time using $\frac{(m+n)\alpha(m,n)}{\log n \log^{(2)} n \log^{(3)} n}$ processors, where $\alpha(m,n)$ is the inverse Ackermann function. Cole and Vishkin (1986b) then designed optimal algorithms for list ranking and prefix sum, and used these results to obtain a slightly faster connected components algorithm that runs in $O(\log n)$ time using $\frac{(m+n)\alpha(m,n)}{\log n}$ processors. The running time of their algorithm is the same as the algorithms of Shiloach and Vishkin (1982) and Awerbuch and Shiloach (1987) but uses fewer processors. Finally, Gazit (1991) presented an optimal randomized algorithm for the Arbitrary CRCW PRAM that runs in $O(\log n)$ time using $O(\frac{m+n}{\log n})$ processors.

Several researchers have considered a more realistic model of parallel computation, the Asynchronous PRAM (APRAM). Gibbons (1989) defined a family of APRAMs that vary in the types of synchronization steps that are permitted. Gibbons measured the cost of accessing the shared memory in terms of the communication delay. In the *Phase* APRAM model, a computation consists of a series of phases, where during a phase processors run asynchronously for a predetermined number of steps. At the end of each phase, all of the processors are synchronized. Gibbons presented Phase APRAM algorithms for several basic problems, including prefix sum, list ranking, and merging.

Martel et al. (1989) presented an APRAM model in which asynchronous processors are randomly assigned to tasks. Thus their asynchronous algorithms can adapt to the availability of processors and are fault tolerant. To measure the performance of an algorithm, they define the *work* of an algorithm to be the total number of single processor operations performed by all processors during the execution of the algorithm. Work is analogous to the processor-time product for PRAM algorithms. Martel et al. (1990) showed that any p-processor CRCW PRAM algorithm can be simulated on an asynchronous CRCW PRAM with $O(p)$ expected work per parallel step using up to $\frac{p}{\log p \log^* p}$ processors.

In the APRAM model of Cole and Zajicek (1989), processors are asynchronous and may access any cell of the shared memory in constant time. They assume that a processor can access a block of memory to read or write in one atomic operation. Since this assumption is not realistic, they consider an algorithm *weak* if it requires atomic access to more than one variable. In their model, a computation is a sequence of all the steps executed by the processors. The computation can be partitioned into *rounds*, where a round is a minimal sequence of steps such that every processor executes at least one step. Cole and Zajicek

presented a complicated weak $O(\log n)$ round graph connectivity algorithm. Later, Cole and Zajicek (1990) broadened their notion of rounds to include probabilistic delays between steps of each processor. During a round, each processor flips a coin to determine whether it should execute an instruction or wait. They presented two APRAM models, a bounded delay model and an unbounded delay model, and showed expected time bounds for prefix sum and list ranking algorithms for each model.

Nishimura (1990) took a different probabilistic approach to the APRAM model and considered probability distributions over the interleaving of processor steps. She showed that the expected maximum number of steps taken by any processor for synchronizing p processors or for list ranking on a list of size p is $O(\log p)$, assuming that the distribution is uniform.

In this chapter we present two deterministic APRAM models. The first is an APRAM that has only atomic **read** and **write** primitives. The second is an APRAM that has limited **read-modify-write** primitives. We then present efficient nonoblivious APRAM connected components algorithms for each model. All of our algorithms use $m + n$ processors. We introduce our approach, which is different from the one used in the synchronous algorithms, in Algorithm I. Algorithm I runs on an APRAM with only atomic **read** and **write** primitives and requires $O(n \log n)$ rounds. Algorithms II and III run on an APRAM with limited **read-modify-write** primitives. Both algorithms require $O(\log n)$ rounds. Algorithm III is more efficient than Algorithm II and uses fewer global synchronizations.

We introduce and motivate our APRAM models in Section 4.2. In Section 4.3, we discuss the design of APRAM algorithms. In Section 4.4, we briefly describe the synchronous connected components algorithms. Although our APRAM algorithms take a

different approach, there are some similarities. We present Algorithm I in Section 4.5, Algorithm II in Section 4.6, and Algorithm III in Section 4.7.

Two problems that are closely related to the connected components problem are finding a spanning forest or a minimum spanning forest of a graph. In Section 4.8, we describe how to modify Algorithms II and III to obtain algorithms for computing a spanning forest and a minimum spanning forest. We give a brief conclusion in Section 4.9.

4.2. Model of Computation

4.2.1. The PRAM

The PRAM is an idealized parallel computer that allows algorithm designers to concentrate on the computational aspects of problems without concern about the reliability and synchronization of processors. A PRAM consists of a set of reliable synchronous processors, each with its own local memory, and a global shared memory through which processors communicate. Each step of a processor of the PRAM consists of three parts, called *stages*. In the first stage, a processor may read a value from one shared memory cell into local memory. In the second stage, a processor may perform a local computation on values in its local memory. In the third stage, a processor may write a value from local memory into one shared memory cell. Each stage requires unit time. Since the processors of the PRAM are synchronous, they all execute the same stage at the same time.

Various models of PRAMs restrict the execution of the read and write stages. In the EREW PRAM, at each step, in the read and write stages at most one processor may read from or write into a particular memory cell. In the CREW PRAM, at each step, in the read stage any number of processors may read from the same memory cell, but in the write stage at most

one processor may write into each memory cell. In the CRCW PRAM, at each step, in the read and write stages any number of processors may read from or write into the same memory cell. If more than one processor writes into a cell, then the value that is written depends on the particular CRCW model.

Three popular CRCW PRAM models are the Priority, Arbitrary, and Common models. For each model, we describe what happens when two or more processors simultaneously try to write into the same cell. In the Priority model, each processor has a unique fixed priority. The processor with the highest priority is the one that succeeds. In the Arbitrary model, some arbitrary processor succeeds, but it is not known *a priori* which one. In the Common model, all processors must write the same value.

Numerous parallel algorithms and computational techniques have been developed for the PRAM. Gibbons and Rytter (1988), Akl (1989), Karp and Ramachandran (1990), and Ja'Ja' (1991) surveyed PRAM algorithms. Fich et al. (1988a), Fich et al. (1988b), Chlebus et al. (1988), and Boppana (1989), among others, studied the relationships among the various PRAM models.

4.2.2. The asynchronous PRAM

Although the PRAM is an important and fundamental model of parallel computation, it is not physically realizable. The PRAM model hides the cost of synchronization. As the number of processors becomes large, it becomes impractical to synchronize the processors using a single global clock. In addition, concurrent accesses to the shared memory must also be synchronized. We present a more realistic model of parallel computation, the APRAM. Our APRAM consists of a set of reliable asynchronous processors, each with its own local

memory, and a global shared memory. No global clock synchronizes the processors, and access to the shared memory is also asynchronous.

As in the PRAM, each step of a processor of the APRAM consists of a read stage, a local computation stage, and a write stage. The difference is that since the processors are asynchronous, some may be reading from the shared memory at the same time others are writing into the shared memory. A *memory operation* is either reading from or writing into one cell of the shared memory. Lamport (1986) considered the problem of concurrent reading and writing of the same cell using safe, regular, and atomic registers. In our APRAM model, we allow at most one memory operation to be performed on a cell of the shared memory at any time, thus each read operation and write operation is atomic. In real machines, hardware can ensure that at all times, only one processor performs a memory operation on a shared memory cell. Since the memory operations performed on each cell are serialized, there is no ambiguity about the value that is written or read.

A *primitive* operation is a sequence of one or more memory operations performed atomically on a single cell of the shared memory. We consider two models of the APRAM that have different kinds of primitive operations. The first is the *basic* APRAM, whose only primitive operations are atomic **read** and **write**. The second is an APRAM with **read-modify-write** primitives. A processor executing a **read-modify-write** primitive atomically reads a value v from a cell c of the shared memory into local memory, performs a local computation that may depend on v and some other values stored in local memory, and then writes a value into cell c. We note that Algorithms II and III actually require only limited **read-modify-write** operations, namely **replace-min** and **increment**, defined as follows.

A processor executing a **replace-min** primitive atomically reads a value v from a cell c of the shared memory, compares v with a value v' in local memory, and then writes v' into c if $v' < v$. A processor executing an **increment** primitive atomically reads a value v from a cell c of the shared memory, increments v by 1, and then writes the value $v+1$ into c. Each primitive can be implemented by a single machine instruction on a conventional computer.

An execution of an algorithm consists of a finite sequence of arbitrarily interleaved stages of the steps taken by all of the processors of the APRAM, with the restriction that when a processor executes a **read-modify-write** primitive, the stages of the step are atomic with respect to the memory cell that is accessed. We can partition the execution of the algorithm into *rounds*, where a round is a minimal sequence of stages such that every processor takes at least one complete step. We measure the running time of our algorithms using rounds. The number of rounds is a fair measure if we assume that all processors run at about the same speed and thus perform about the same number of operations. This assumption is reasonable because real parallel computers usually consist of identical processors. The algorithms must be correct, however, regardless of the speeds of the processors.

4.3. The Design of APRAM Algorithms

Ideally we would like to design algorithms for which the execution of each processor is independent of the executions of other processors. In that case, no processors impede the progress of other processors. But most algorithms, and nonoblivious algorithms in particular, require processors to communicate data to each other. Thus synchronization between

processors is needed. In Section 4.3.1 we review previous work in wait-free data structures, and in Section 4.3.2 we discuss methods for synchronizing processors.

4.3.1. Wait-free data structures

An implementation of a parallel data structure in shared memory is *wait-free* if every processor is guaranteed to complete an operation on the data structure in a finite number of steps regardless of the speeds of the other processors. A wait-free implementation tolerates failures of processors because the failure of other processors does not prevent the completion of an operation.

Herlihy (1988) defined a hierarchy of data structures such that at each level no data structure has a wait-free implementation in terms of data structures from lower levels. For example, Herlihy shows that using only atomic **read** and **write** registers, it is impossible to construct wait-free implementations of queues and stacks, and synchronization primitives such as **test-and-set** and **fetch-and-add**. A primitive is *universal* if wait-free data structures can be constructed from the primitive. Herlihy established the existence of universal primitives. It is not clear, however, that universal primitives can be implemented in hardware.

Some researchers have designed wait-free algorithms. Aspnes and Herlihy (1990) considered an APRAM model with only shared atomic registers and gave an algebraic characterization of a class of data structures that have wait-free implementations. Recently, Anderson and Woll (1991) designed a wait-free algorithm for the union-find problem.

4.3.2. Processor synchronization

A *barrier synchronization* is a sequence of steps that synchronizes all processors. Every processor executing the barrier synchronization must complete the operation before any processor may proceed further. Barrier synchronization may be used to ensure that all processors have reached a particular point of the computation before continuing.

On a basic APRAM, a barrier synchronization can be implemented with a binary tree. In the binary tree method, each processor is associated with one leaf in the tree, and each internal node of the tree is associated with some processor. A processor executing a barrier synchronization marks its corresponding leaf. A processor corresponding to an internal node x marks x after both children of x have been marked. If some child of x is not marked, then the processor waits for a while and checks again. The barrier synchronization completes when the root is marked. On a basic APRAM with p processors, the binary tree method for barrier synchronization requires exactly $\lceil \log_2 p \rceil$ rounds since the processor corresponding to the root must wait for $\lceil \log_2 p \rceil - 1$ levels of nodes to be marked.

Barrier synchronization can be made more efficient if processors do not have to wait to execute the barrier. The results of Fischer et al. (1985), Loui and Abu-Amara (1987), Chor et al. (1987), and Herlihy (1988) can be used to show that there is no wait-free implementation of barrier synchronization on an APRAM that has only atomic **read** and atomic **write** operations. Thus we consider an APRAM that can read and write a cell in one atomic operation, i.e., an APRAM with **read-modify-write** primitives.

On an APRAM with an **increment** primitive, barrier synchronization can be implemented using a counter initialized to 0. In the counter method, a processor executing a

barrier synchronization increments the counter. On an APRAM with p processors, the barrier synchronization completes when the value of the counter reaches p. With the counter method, a processor does not have to check whether other processors have executed the barrier synchronization before incrementing the counter. Thus barrier synchronization requires only one round. Note that processors must still wait for the barrier synchronization to complete before they can proceed further in the algorithm.

A straightforward way to obtain an APRAM algorithm is to take a PRAM algorithm and insert a barrier synchronization after each step of the PRAM algorithm. This conversion is inefficient, however. For a basic APRAM with p processors, the round complexity of the APRAM algorithm is about $\log_2 p$ times greater than the round complexity of the corresponding PRAM algorithm. For an APRAM with an increment primitive, the round complexity of the APRAM algorithm is about 2 times greater than the PRAM algorithm. In either model, with so many barrier synchronizations, the APRAM algorithm proceeds no faster than the rate of the slowest processor at each step. Fast processors must wait for slow processors.

One way to reduce the cost of synchronization is to synchronize only a constant number of processors at each synchronization point. For example, on a basic APRAM, if only a constant number of processors need to be synchronized at each step, then the APRAM algorithm is slowed by only a constant factor. Cole and Zajicek (1989) showed that for a class of algorithms in which the communication pattern is oblivious, each synchronization point involves only a constant number of processors. These algorithms include Batcher's bitonic sort (Batcher, 1968), parallel summation, and prefix sum. The memory cells that are used in these algorithms can be treated as an implicit complete binary tree. During the

execution of an algorithm, a processor associated with an internal node may proceed once both of its children have been computed. Note that each node is written by only one processor.

Many oblivious PRAM algorithms, and especially tree-based PRAM algorithms, can be converted into APRAM algorithms in this manner since the memory cells that are to be accessed and the number of processors that need to be synchronized at each step can be determined a priori. Other algorithms that fall into this category include evaluation of algebraic expressions, evaluation of associative binary functions, and numeric computations such as matrix multiplication.

For nonoblivious algorithms, however, as is the case for many graph algorithms, efficient synchronization is more difficult. Since the communication pattern depends upon the input, an algorithm cannot determine a priori the cells that each processor will access. Thus each barrier synchronization must involve every processor.

In this chapter we present efficient nonoblivious APRAM algorithms for computing connected components, spanning forest, and minimum spanning forest. In our connected components APRAM algorithms, to reduce waiting, we introduce a variation of barrier synchronization, which we call *rolling synchronization.* We describe rolling synchronization.

Rolling synchronization can be implemented with the same data structures used for barrier synchronization. We use two methods of rolling synchronization: the tree method and the counter method. Suppose we want to synchronize all of the processors after they have completed some set of steps L of a loop. After completing L, processor p tries to execute its part of the rolling synchronization by either marking the nodes of a tree or incrementing a counter as in barrier synchronization. The difference is that if some processor

has not yet executed its part of the rolling synchronization, then p executes L again. Then p tries to execute its part of the rolling synchronization, if p has not already completed it, and checks again whether all processors have completed the rolling synchronization. Processor p repeatedly executes L until every processor has completed the rolling synchronization. The rolling synchronization completes after every processor has executed L at least once. Rolling synchronization is an improvement over barrier synchronization since performing the steps L again may do more useful work. The algorithm must be correct, however, for an arbitrary number of iterations of L.

In real machines, processor synchronization can be expensive. Axelrod (1986) and Dubois and Briggs (1991), among others, have examined the degradation of the performance of multiprocessor machines caused by barrier synchronizations. Thus, in our APRAM algorithms, we attempt to minimize the number of synchronization points.

4.4. Synchronous Algorithms

4.4.1. Connected components algorithms

Shiloach and Vishkin (1982) and Awerbuch and Shiloach (1987) presented similar connected component algorithms. We briefly describe their methods because our algorithms use some of their ideas. In our discussion, we will focus on the algorithm of Awerbuch and Shiloach since it is simpler.

Assume that the vertices of G are numbered from 1 to n. Then the number of a vertex is its *id*. In our discussion, we will refer to vertices by their *id*'s. The notation $u < v$ means that the *id* of vertex u is smaller than the *id* of vertex v.

The synchronous algorithm uses an Arbitrary CRCW PRAM. Each edge of G is represented by two oppositely directed edges. The algorithm assigns a processor $p(i,j)$ to each edge (i,j) and a processor $p(v)$ to each vertex v. Thus, the number of processors required by the algorithm is $2m + n$.

Each vertex i has a *parent* $P(i)$. If a vertex is a root, then its parent is itself. The *grandparent* of i is the parent of $P(i)$.

A *rooted tree* is a tree whose edges are directed toward the *root.* A *star* is a rooted tree of height 1. The parent-child relation defines a directed graph called the *parents graph, PG*, which is a forest of rooted trees; PG has the same vertices as G. We will call an edge of PG an *arc* to distinguish the edges of PG from the edges of G. The arcs of PG are $(i,P(i))$, for all vertices i.

Throughout the execution of the algorithm, PG is a forest of rooted trees with self-loops at the roots. The set of vertices of each tree of PG is a subset of vertices of some connected component of G. The vertices of two trees of PG belong to the same connected component if there is an edge of G with an endpoint in each tree. The processor corresponding to such an edge tries to combine two such trees by *hooking* one tree to the other. The algorithm hooks a tree T_1 to a tree T_2 by making some vertex of T_2 the parent of the root of T_1.

After the hooking operation, the algorithm reduces the height of each tree with a *shortcut* operation, in which each vertex takes its grandparent to be its new parent. The algorithm terminates when no trees in PG can be combined, and every tree is a star. All vertices that belong to the same connected component have the same parent in PG.

Arbitrary CRCW PRAM Connected Components Algorithm

Initialization

For each processor $p(i)$

$P(i) := i$;

For each processor $p(i,j)$

loop

Part 1: (Conditional star hooking)
if i belongs to a star **and** $P(i) > P(j)$ **then**
$P(P(i)) := P(j)$;
endif

Part 2: (Stagnant star hooking)
if i belongs to a star **and** $P(i) \neq P(j)$ **then**
$P(P(i)) := P(j)$;
endif

Part 3: (Shortcut operation)
if i does not belong to a star **then**
$P(i) := P(P(i))$;
else
stop;
endif

end loop

A processor determines whether a vertex belongs to a star by using procedure Star_Check. At the termination of Star_Check, $ST(i)$ is **true** if i belongs to a star and **false** otherwise.

Procedure Star_Check

For each processor $p(i)$

$ST(i) :=$ **true**;

```
if P(i) ≠ P(P(i)) then
    ST(i) := false;
    ST(P(P(i))) := false;
endif
ST(i) := ST(P(i));
```

In Part 1, processors that correspond to edges leaving a star try to hook the star to another tree, which may also be a star. If more than one processor tries to hook the star, then an arbitrary processor succeeds. If v is a vertex of a star that is hooked, then at the end of Part 1, v is no longer a vertex of a star. A star that has not been hooked to another tree or hooked onto by another star is *stagnant*. In Part 2, stars that are stagnant after Part 1 are hooked to trees. Stars that are stagnant after Part 2 correspond to connected components that are complete. Thus, a processor corresponding to an edge of such a star stops. In Part 3, the algorithm reduces the height of each tree that is not a star with a shortcut operation. If the height of a tree is h, then after a shortcut operation the height of the tree is $\lceil h/2 \rceil$. The height of every tree whose height is at least 2 is reduced by a factor of at least 3/2 because in the worst case a shortcut operation reduces a tree whose height is 3 to a tree whose height is 2.

The main difference between the algorithm of Shiloach and Vishkin and the algorithm of Awerbuch and Shiloach is that in the former algorithm, during the hooking operation, trees that are not stars may also be hooked. In Part 1, if i is a root or a child of a root of a tree and $P(i) > P(j)$, then processor $p(i,j)$ tries to hook the tree containing vertex i to $P(j)$.

Shiloach and Vishkin (1982) and Awerbuch and Shiloach (1987) established the correctness of their algorithms. We briefly verify the running time. Consider each iteration of the three Parts. Parts 1 and 2 ensure that if a star does not contain all of the vertices of

some connected component, then the star is either hooked to another tree or hooked onto by another star to yield a new tree with height at least 2. Part 3 reduces the height of every tree with a height of at least 2 by a factor of at least 3/2 with a shortcut operation. The algorithm ensures the progress of Part 3 by avoiding the creation of cycles during the hooking steps. In Part 1, the roots of stars are hooked only to vertices with smaller *id*'s, and in Part 2, only the roots of stagnant stars are hooked.

For each connected component C, during each iteration of the algorithm, each star that contains vertices of C is hooked to another tree or hooked onto by another star to form a new tree of height at least 2. After a shortcut operation, the height of the new tree is reduced by a factor of at least 3/2. Thus, the sum of the heights of the trees in PG that contain the vertices of C decreases by a factor of at least 3/2. After $O(\log n)$ iterations, the vertices of C belong to a single star. Since each iteration comprises O(1) steps, the algorithm requires $O(\log n)$ total steps.

4.4.2. Spanning forest algorithm

If C is a connected component of G, then a *spanning tree* of C is a tree that is a subgraph of G that contains every vertex of C. A *spanning forest* of G is a set of trees consisting of a spanning tree for each connected component of G. We can modify the connected components algorithm to find a spanning forest of G by keeping track of the processors that perform the hooking operations. We introduce a variable $Proc(i)$ for each vertex i. Initially $Proc(i) =$ **nil**. Parts 1 and 2 of the connected components algorithm are modified as follows.

Part 1: (Conditional star hooking)
if i belongs to a star **and** $P(i) > P(j)$ **then**
$P(P(i)) := P(j)$;
if $P(P(i)) = P(j)$ **then**
$Proc(P(i)) := p(i,j)$;
endif
endif

Part 2: (Stagnant star hooking)
if i belongs to a star **and** $P(i) \neq P(j)$ **then**
$P(P(i)) := P(j)$;
if $P(P(i)) = P(j)$ **then**
$Proc(P(i)) := p(i,j)$;
endif
endif

In Parts 1 and 2, if a vertex x is the root of a tree and x is hooked to a vertex y, then the processors that may have set $P(x)$ to y try to write their names into $Proc(x)$, and an arbitrary one succeeds. Note that if more than one processor tried to hook x to y, then the processor whose name is written in $Proc(x)$ might not be the processor that hooked x to y. The algorithm remains correct, however, since $Proc(x)$ is the name of a processor that corresponds to an edge of G whose endpoints are in the trees of PG that contain x and y. Note that if a vertex r is the root of a star, then $Proc(r) =$ **nil**. After all vertices of each connected component are contained in a single star, the processors whose names are stored in $Proc(i)$, for $i = 1, \ldots, n$, correspond to the edges of a spanning forest of G.

4.4.3. Minimum spanning forest algorithm

A graph G is a *weighted graph* if a weight $w(i,j)$ is associated with each edge (i,j) of G. If G is a weighted graph, then a *minimum spanning forest* of G is a spanning forest of G such that the sum of the weights of the edges in the trees of the spanning forest is a minimum.

Awerbuch and Shiloach (1987) also presented a minimum spanning forest algorithm for the Priority CRCW PRAM that runs in $O(\log n)$ time using $O(m+n)$ processors.

The minimum spanning forest algorithm is similar to the connected components algorithm, except that during hooking operations, the processor corresponding to the edge of least weight leaving a star hooks the star. To determine the edge of least weight, the algorithm assigns processors to edges such that the smaller the weight of an edge, the higher the priority of the corresponding processor. Thus, if more than one processor tries to hook a star, then the processor corresponding to the edge of least weight is the one that succeeds. As in the spanning forest algorithm, a variable for each vertex is used to keep track of the processor that successfully performed the hooking operation. After all vertices of each component are contained in a single star, the processors that successfully hooked stars correspond to the edges of a minimum spanning forest of G.

4.5. APRAM Connected Components Algorithm I

The main difficulty with an asynchronous implementation of the synchronous algorithm is that in the asynchronous case, without a barrier synchronization after Part 1, a processor executing Part 2 cannot determine which stars are stagnant. This is because it does not know which other processors are performing hooking operations. We show how a cycle may be created.

In Part 2 of the synchronous algorithm, since only stagnant stars are hooked to other trees, no cycles are created. In an asynchronous implementation without barrier synchronization, however, some processors may be executing different Parts of the algorithm. Let r be the root of a star S and r' the root of another star such that $r' > r$. Suppose a

processor executing Part 2 determines that S is stagnant and hooks r to r'. If at the same time a different processor executing Part 1 hooks r' to r, then a cycle of length 2 is created. In general, longer cycles can be created. If there are cycles, then the progress of the shortcut operations is not guaranteed.

In our algorithms, to avoid creating cycles, a vertex is hooked only to a vertex with a smaller *id*. In Algorithm I, the hooking operations act on individual vertices rather than trees, as in the synchronous algorithm. Let (x, y) be an edge of G. Let u be the parent of x and v be the parent of y. Note that u may be x and v may be y. The parents graph PG is the same as in the synchronous algorithm of Section 4.4. In PG, if u and v are different, then $\{u,v\}$ is an *eligible pair*. Suppose $u < v$. Then processor $p(x,y)$ hooks v to u, and vertices u and v are *coupled*. After the hooking step, a processor performs a shortcut operation on each endpoint.

We now present Algorithm I, which runs on a basic APRAM. We use uppercase names for variables that are stored in the shared memory and lowercase names for variables stored in each processor's local memory. Each vertex v has a parent $P(v)$. Unlike the synchronous algorithm, we use one processor for each edge of G. A processor corresponding to an edge has local variables *parent*() and *grandparent*() for each endpoint to hold the endpoint's parent and grandparent, respectively.

APRAM Connected Components Algorithm I

Initialization

For each processor $p(v)$

$P(v) := v$;

```
For each processor p(x,y)

while there are eligible pairs

Part 1: (Vertex hooking)
    parent(x) := P(x);
    parent(y) := P(y);

    if (parent(x) ≠ parent(y)) then
        if parent(x) > parent(y) then
            P(parent(x)) := parent(y);
        else
            P(parent(y)) := parent(x);
        endif
    endif

Part 2: (Shortcut operation)
    parent(x) := P(x);
    grandparent(x) := P(parent(x));
    P(x) := grandparent(x);

    parent(y) := P(y);
    grandparent(y) := P(parent(y));
    P(y) := grandparent(y);

    rolling synchronization with binary tree;

endwhile
```

For now we assume that the algorithm can determine whether there are eligible pairs. We will justify this assumption shortly. In Part 1, processors that correspond to edges whose endpoints have different parents hook the parent with the larger *id* to the other parent. Note that a vertex may be hooked to another vertex of the same tree in PG. In Part 2, processors perform a shortcut operation on each endpoint. Processors iterate Parts 1 and 2 until no eligible pairs remain in PG. Then all the vertices in each connected component have the same parent.

We show that the algorithm correctly finds the connected components of G. We can establish by induction on the number of hooking operations that all vertices of a tree of PG are in the same connected component. Initially, each tree of PG comprises a single vertex. We verify that a hooking operation couples only vertices that belong to the same connected component.

In PG, if a vertex u of a tree is coupled with a vertex v of a different tree, then the processor $p(x,y)$ that performs the hooking operation corresponds to an edge (x,y) of G. Let $P(x)=u$ and $P(y)=v$. By the inductive hypothesis, vertices x and u belong to the same component and vertices y and v belong to the same component. Vertices x and y belong to the same component since they are joined by an edge of G. Thus vertices $u, v, x,$ and y all belong to the same connected component, and the hooking operation is correct.

Since the only primitive operations allowed on the shared variables are atomic **read** and **write**, the update of $P(v)$ by one processor may overwrite that of another. But the algorithm maintains the invariant that $P(v)$ is always a vertex in the same connected component as v. In the hooking operation, v is hooked only to vertices that belong to the same connected component as v. In the shortcut operation, at the time a processor writes $P(v)$, it is possible that $grandparent(v)$ is no longer the grandparent of v because $P(parent(v))$ may have been updated. But the algorithm sets $P(v)$ to a vertex in the same connected component as v because at the time $grandparent(v)$ was determined, the algorithm learned that v and $grandparent(v)$ are in the same connected component.

Next we show that when the algorithm terminates, if two vertices belong to the same connected component, then they are in the same tree. Suppose x and y are two vertices that belong to the same connected component. Then there is a path W from x to y in G. If x and

y are in different trees of PG, then the endpoints of some edge e in W must belong to different trees. Since the parents of e form an eligible pair, the algorithm makes another iteration.

To analyze the performance of the algorithm, we partition the execution of the algorithm into *phases*, where a phase is a minimal sequence of the stages of the steps of the processors in which every processor completes at least one iteration of the **while** loop. Note that our definition of a phase is not the same as the definition of Gibbons (1989). Although the ideas of phases and rounds are similar, except in scale, their boundaries do not necessarily coincide because a stage of a step of a processor that completes a phase might not complete a round. We can, however, bound the number of rounds in a phase. If an iteration of the **while** loop consists of i APRAM processor steps, then a phase contains at most i rounds since in each round every processor takes at least one step. We review the definitions of stage, step, round, and phase in Table 4.1.

During a phase, if a vertex v is hooked to more than one vertex (in sequence), then the last processor to update $P(v)$ determines the vertex to which v is hooked at the end of the phase. Since v may be hooked only to a vertex that has a smaller *id*, if v is hooked to another vertex during a phase, then at the end of the phase $P(v)$ has decreased by at least 1 since the beginning of the phase. The algorithm maintains the invariant that from one phase to the next, $P(v)$ never increases.

Theorem 4.5.1: Algorithm I requires at most $n - 1$ phases.

Proof: Let the connected component C contain n_c vertices; without loss of generality, assume that the vertices are numbered 1 through n_c. During each phase, the vertex w of C with the largest $P(w)$ is hooked to some vertex. Thus, at the end of the phase, $P(w)$ has

Table 4.1. A summary of the definitions of stage, step, round, and phase.

Stage	One read operation, local computation, or write operation executed by one processor.
Step	Each step of a processor comprises one read stage, one local computation stage, and one write stage.
Round	A minimal sequence of stages such that every processor completes at least one step.
Phase	A minimal sequence of stages such that every processor completes at least one iteration of a loop.

decreased by at least 1. At the end of phase k, for each vertex v of C, $P(v) \leq n-k$. Thus, $P(v) = 1$ for every vertex v after at most $n_c - 1$ phases. Since $n_c \leq n$, the result follows. □

If G is a star with n vertices and the *id* of the root is n, then there is an execution of Algorithm I on G that requires $n-1$ phases. At the end of phase k, the $n-k$ trees of PG consist of a star rooted at vertex $n-k$ with children n through $n-k+1$ and isolated vertices 1 through $n-k-1$.

Finally, we discuss how the algorithm determines whether there are eligible pairs. Synchronization is needed to ensure that every processor has had an opportunity to find any eligible pairs during the current phase. Otherwise a processor may terminate prematurely.

In Algorithm I, a shared variable EP is used to keep track of whether there are eligible pairs; EP is initially **true**. At the end of each phase, if some processor found an eligible pair

during the latest phase, then *EP* is set to **true**. But if no processor found an eligible pair, then *EP* is set to **false**, and the algorithm terminates.

The determination of whether some processor found an eligible pair during a phase is performed in conjunction with the rolling synchronization. The binary tree used for rolling synchronization is also used to keep track of whether some processor has found an eligible pair. When a processor marks its corresponding leaf in the synchronization tree, the processor marks its leaf **true** if it found an eligible pair during the phase, and **false** if it did not. A processor marks an internal node **true** if either child is marked **true** and **false** otherwise. The processor corresponding to the root of the synchronization tree sets *EP* for the next phase before marking the root to signal the completion of the rolling synchronization for the current phase. If any processor found an eligible pair during the previous phase, then *EP* is set to **true**.

Each processor checks *EP* before executing an iteration of the **while** loop. The algorithm terminates after the first phase during which no processor finds an eligible pair. The determination of the existence of eligible pairs adds one more phase to the algorithm.

On a basic APRAM, rolling synchronization can be implemented with a binary tree. For simplicity, a separate tree can be used for each phase of the algorithm. To reduce space, however, two synchronization trees can be used alternately as follows. For either tree, a processor that corresponds to an internal node first erases the marks of the children of the node and then marks the node. At the time the root of the synchronization tree is marked, the marks for all other nodes of the tree are erased.

To be able to erase the mark on the root of the synchronization tree, two trees are needed. If the mark on the root is erased too early, then not every processor would see the

mark. If the mark is erased too late, then a fast processor may see the mark and complete the next phase before the mark is erased. With two trees, one tree is used for odd phases and one tree for even phases. Before a processor corresponding to the root of tree marks the root, it erases the marks on the children of the root and also the mark on the root of the other tree. With two synchronization trees, at most one root is marked at any time, and no mark on a root is erased until every processor has had an opportunity to see it.

Theorem 4.5.2: Algorithm I requires $O(n \log n)$ rounds.

Proof: Algorithm I requires at most $n-1$ phases. Since each rolling synchronization requires $O(\log m) = O(\log n)$ rounds, each phase of the algorithm requires $O(\log n)$ rounds. Thus the total number of rounds required by the algorithm is $O(n \log n)$. □

We note that the straightforward asynchronous implementation of the synchronous algorithm of Section 4.4 yields an APRAM algorithm that requires $O(\log^2 n)$ rounds since the number of phases is $O(\log n)$. The main purpose of Algorithm I, however, is to present our asynchronous approach.

4.6. APRAM Connected Components Algorithm II

4.6.1. The algorithm

Algorithm I is inefficient because if during a phase a vertex v may be hooked to several different vertices, then in the worst case at the end of each phase v is hooked to the vertex whose *id* is the largest. Algorithm II runs on an APRAM with two **read-modify-write** primitives, **replace-min** and **increment**. Using these stronger primitives, we can design a more efficient algorithm. The **replace-min** primitive ensures that during each hooking and shortcut operation, if the algorithm assigns v a new parent, then the *id* of the new parent is

smaller than the *id* of the old parent. The **increment** primitive allows each rolling synchronization to be executed with a counter in one round.

Algorithm II differs from Algorithm I in that the algorithm hooks entire trees, and only by their roots, to other trees. This restriction is necessary for the analysis of the algorithm. We use the notation **replace-min**<<*s*>> to denote that the assignment statement *s* is executed atomically using a **replace-min** primitive.

APRAM Connected Components Algorithm II

Initialization

For each processor $p(v)$

$P(v) := v$;

For each processor $p(x,y)$

while there are eligible pairs

Part 1: (Shortcut operation)
parent$(x) := P(x)$;
grandparent$(x) := P($*parent*$(x))$;
replace-min<<$P(x) :=$ **min** $\{P(x),$ *grandparent*$(x)\}$>>;

parent$(y) := P(y)$;
grandparent$(y) := P($*parent*$(y))$;
replace-min<<$P(y) :=$ **min** $\{P(y),$ *grandparent*$(y)\}$>>;

rolling synchronization with counter;

Part 2: (Tree hooking)
parent$(x) := P(x)$;
parent$(y) := P(y)$;

if *parent* $(x) \neq$ *parent* (y) **then**
if (x was a root at the beginning of the phase) **and** (*parent* $(x) >$ *parent* (y)) **then**
replace-min<<P (*parent* (x)) := **min** {P (*parent* (x)), *parent* (y)}>>;
else if (y was a root at the beginning of the phase) **and** (*parent* $(y) >$ *parent* (x)) **then**
replace-min<<P (*parent* (y)) := **min** {P (*parent* (y)), *parent* (x)}>>;
endif
endif

if (x was a root at the beginning of the phase) **then**
parent $(x) := P(x)$;
grandparent $(x) := P$ (*parent* (x));
replace-min<<$P(x)$:= **min** {$P(x)$, *grandparent* (x)}>>;
endif

if (y was a root at the beginning of the phase) **then**
parent $(y) := P(y)$;
grandparent $(y) := P$ (*parent* (y));
replace-min<<$P(y)$:= **min** {$P(y)$, *grandparent* (y)}>>;
endif

rolling synchronization with counter;

Part 3: (Shortcut operation)
Same as in Part 1;

rolling synchronization with counter;

endwhile

We give an example to demonstrate the improved performance of Algorithm II. Suppose G is a star and vertex n is the root. In the worst case, Algorithm I requires $n-1$ phases. In Algorithm II, at the end of the first phase, the parent of vertex n is vertex 1. At the end of the second phase, vertex 1 is the parent of every vertex. Thus Algorithm II requires at most two phases.

4.6.2. Analysis

The correctness of Algorithm II can be established in a manner similar to that used for Algorithm I. Thus we will analyze only the performance. Let u and v be vertices such that $u < v$. In Algorithm II, a processor p tries to hook v to u if $\{u, v\}$ is an eligible pair and v was the root of a tree at the beginning of the current phase. Note that at the time processor p tries to hook v to u, vertex v may no longer be the root of a tree since v may have been previously hooked onto some other vertex.

During a phase, if on the last time v is hooked onto a vertex, v is hooked onto u, then during that phase u *adopts* v, v is *adopted by* u, and u and v *participate in an adoption.* During a phase, a vertex x is *stagnant* if x does not participate in an adoption. Note that x may be stagnant even if some vertex y is hooked to x since during the phase, y may subsequently be hooked to some other vertex. If during the phase v is adopted by u, then the tree rooted at v is adopted by the tree containing u, and the trees are *merged.* In our discussion, the adoption of the root of a tree implies the merging of trees.

The rolling synchronization after Part 2 ensures that entire trees are adopted. The algorithm may perform shortcut operations on the root of a tree after the root is hooked to another vertex. The algorithm performs shortcut operations on the remaining vertices of the tree only after the root has been adopted during the current phase. If there were no rolling synchronization after Part 2, then through shortcut operations, some vertex of the tree might take on a parent that was only a temporary parent of the root, and the algorithm would not be guaranteed to merge entire trees.

Let C be a connected component of G and let $V(C)$ be the vertices of C. Let u and v be two vertices of $V(C)$.

Lemma 4.6.1: If $\{u, v\}$ is an eligible pair, $u < v$, and v is the root of a tree at the beginning of phase k, then at the end of Part 2 of phase k, vertex v is adopted.

Proof: Let (x, y) be an edge of G such that $P(x) = u$ and $P(y) = v$. Then processor $p(x,y)$ finds the eligible pair $\{u, v\}$. Throughout the computation, $P(v)$ decreases monotonically. In Part 2 of phase k, processor $p(x,y)$ hooks v to u unless v has already been hooked to another vertex and $P(v) < u$. Thus, at the end of Part 2 of phase k vertex v is adopted by some vertex. □

Lemma 4.6.2: If $\{u, v\}$ is an eligible pair, $u < v$, and u and v are both roots of trees at the beginning of Part 2 of phase k, then at the end of Part 2 of phase $k+1$, vertices u and v have each participated in at least one adoption, and at least one of u and v is adopted.

Proof: Let $p(x,y)$ be a processor such that $P(x) = u$ and $P(y) = v$. Then processor $p(x,y)$ finds the eligible pair $\{u, v\}$. By Lemma 4.6.1, during phase k vertex v is adopted by some vertex. If v is adopted by u, then vertex u also participates in an adoption. If vertex u remains stagnant during phase k, then at the end of Part 2 of phase k, the parent of v must have been a vertex b, where $b < u$. At the end of phase k, after processor $p(x,y)$ performs at least one shortcut operation, the parent of y is a vertex a, where $a \leq b$. During phase $k+1$ processor $p(x,y)$ finds the eligible pair $\{a,u\}$. Since $a < u$ and u is the root of a tree, by Lemma 4.6.1, vertex u is adopted at the end of Part 2 of phase $k+1$. □

Two trees T_1 and T_2 of PG are *adjacent* if there exist vertices x and y such that x is a vertex of T_1, y is a vertex of T_2, and (x, y) is an edge of G. A tree is *tall* if its height is at least 2. A tree is tall if and only if it is not a star. As a consequence of Lemma 4.6.2, we have the following corollary.

Corollary 4.6.3: If u and v are the roots of adjacent stars at the beginning of Part 2 of phase k, then at the end of Part 2 of phase $k+1$, vertices u and v have each participated in at least one adoption, and at least one of u and v is adopted.

Corollary 4.6.4: If u is the root of a star at the beginning of phase k and u is stagnant during phases k and $k+1$, then at the beginning of phase k, every tree adjacent to the star with root u is a tall tree of height at least 3.

Proof: Suppose to the contrary that at the beginning of phase k the height of some tree adjacent to the star rooted at u were at most 2. After a shortcut operation in Part 1, the tree would be a star. But then by Corollary 4.6.3, at the end of Part 2 of phase $k+1$, vertex u would have participated in at least one adoption. □

4.6.2.1. The potential function

To show that the vertices of $V(C)$ form a single star after $O(\log n)$ phases, we measure the progress of the algorithm with a potential function. We first give some intuition behind the potential function. In the synchronous algorithms of Section 4.4, the progress of the algorithms is measured by the sum of the heights of the trees in PG. During each iteration every star participates in an adoption to form a new tree of height at least 2. After a shortcut operation, the height of each tall tree is reduced by a factor of at least 3/2. Thus the sum of the heights of the trees that contain the vertices of $V(C)$ decreases by a factor of at least 3/2.

We show that in Algorithm II, during two consecutive phases, only the sum of the heights of tall trees and the sum of the heights of stars that are adjacent to other stars are guaranteed to decrease by at least a constant factor. During each phase, the algorithm

performs at least two shortcut operations. Thus during two consecutive phases, the algorithm performs at least four shortcut operations.

If the height of a tall tree is h, then after four shortcut operations, the height of the tree is at most $\lceil \lceil \lceil \lceil h/2 \rceil /2 \rceil /2 \rceil /2 \rceil = \lceil h/16 \rceil$. If $2 \leq h \leq 16$, then after four shortcut operations, the height of the tree is reduced to 1. Since $h \geq 2$, after two consecutive phases, the height of the tree decreases by a factor of at least 2.

If $h \geq 17$, then after two consecutive phases, the height of the tree is at most $h/16 + 1$. Since $h \geq 17$, the height of each tree decreases by a factor of at least 272/33.

For stars that are adjacent to other stars, by Corollary 4.6.3, after hooking operations in two consecutive phases, at least half of those stars are adopted to form tall trees. After each hooking operation, the algorithm performs at least one shortcut operation. Since the height of each tall tree is reduced by a factor of at least 3/2, the sum of the heights of those stars that are hooked to form the tall trees is also reduced by a factor of at least 3/2. Stars that are adjacent only to tall trees may remain stagnant for several phases, however. But after a tall tree becomes a star, then all of the stars that were adjacent to the tall tree participate in an adoption within two phases.

The potential function we use corresponds roughly to the sum of the heights of the trees. The difference is that if several stars are adjacent to one tree, then not all of the stars may be counted. This is because the maximum height of the new tree that can be formed by merging all of the stars with the tree is bounded. We will explain this shortly.

Consider the vertices $V(C)$ of C. The vertices of $V(C)$ are partitioned into trees in PG. At the beginning of phase k, let the forest F_k be the set of trees in PG that contain the vertices of $V(C)$. During phase k, the algorithm may merge trees of F_k. The *potential* of the

forest, which we define precisely below, is an upper bound on the sum of the heights of the trees that can be formed if all of the trees in F_k participate in at least one adoption during the next two phases. If the potential of F_k is 1, then the vertices of $V(C)$ are contained in a single star. We will show that after every two phases, the potential of the forest in PG that contains the vertices of $V(C)$ decreases by at least a constant factor. Thus the vertices of $V(C)$ are contained in a single star after O(log n) phases.

We define the potential function by considering the trees in a forest F. Without loss of generality, assume that there are no isolated stars in F, since each such star corresponds to a connected component all of whose vertices are contained in a single star. A tree in F is *external* if it is adjacent to one other tree in F and *internal* if it is adjacent to at least two other trees in F. We will partition F into two subsets, and then define the potential of each subset. The potential of F is the sum of the potentials of the two subsets.

To help clarify the derivation of the potential function, we will use an example. Suppose F is the forest of trees in PG shown in Figure 4.1. Tall trees are represented by the tall triangles, and stars are represented by the short triangles. The dashed lines indicate adjacent trees. For each pair of adjacent trees, there is at least one edge of G between the trees. During our discussion, we will refer to Figure 4.1.

A tree T_1 *covers* a tree T_2 if T_1 is adjacent to T_2. Let T be a tree of F. A *cluster* of T is a set of stars covered by T. A cluster may contain from none to all of the stars covered by T. Call T the *core* of the cluster. Note that T is not in the cluster.

Let AS be the set of stars of F that are adjacent to at least one other star. Let $St(F)$ be a minimum cardinality set of stars of AS such that for every pair of adjacent stars in AS, at least one of the stars is in $St(F)$. Note that $St(F)$ might not be unique. Also, each star in

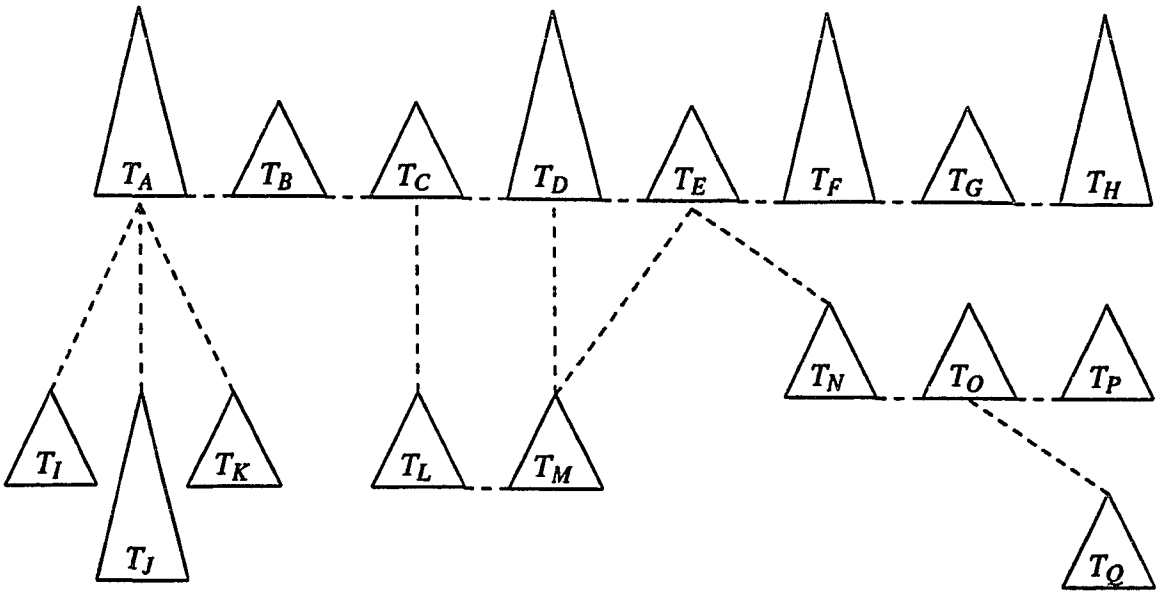

Figure 4.1. An example of a forest F.

$St(F)$ covers at least one star not in $St(F)$; otherwise, $St(F)$ would not be a set of minimum cardinality.

Let $Ta(F)$ be the set of tall trees in F. Let $Co(F) = Ta(F) \cup St(F)$. Finally, let $Cl(F)$ be the set of stars of F that are not in $Co(F)$. An *assignment* is a function that assigns each star S of $Cl(F)$ to a tree of $Co(F)$ adjacent to S. For an assignment A, $A(S)$ is the tree to which star S is assigned. Note that each external star may be assigned to only one tree. We will consider particular assignments later.

Fix an assignment. For each T in $Co(F)$, let $Cl(T)$ be the set of stars of $Cl(F)$ assigned to T. If T is a tall tree, then $Cl(T)$ is a *tall tree cluster*. If T is a star, then $Cl(T)$ is a *star cluster*. It is clear that each tree of F is in exactly one of the subsets $Co(F)$ and $Cl(F)$.

In the example of Figure 4.1, $Ta(F) = \{T_A, T_D, T_F, T_H, T_J\}$. One possible set $St(F)$ of minimum cardinality is $St(F) = \{T_B, T_E, T_L, T_O\}$. For this $St(F)$, one possible assignment yields the clusters $Cl(T_A) = \{T_I, T_K\}$, $Cl(T_B) = \{T_C\}$, $Cl(T_E) = \{T_N\}$, $Cl(T_H) = \{T_G\}$, $Cl(T_L) = \{T_M\}$, and $Cl(T_O) = \{T_P, T_Q\}$.

Lemma 4.6.5: During the execution of Algorithm II, if two trees of F are merged, then at least one of the trees is a tree of $Co(F)$.

Proof: If either tree is a tall tree, then at least one tree is a tree of $Ta(F)$. If both trees are stars, then by definition, at least one tree is a star of $St(F)$. □

We now define the potential of each subset. We define the potential such that the potential of F is an upper bound on the sum of the heights of the trees that can be formed if every tree of F participates in at least one adoption during the next two phases.

Let $ht(T)$ denote the height of a tree T. If T is a tree of $Co(F)$, then define $\Phi(T)$, the potential of T, to be $ht(T)$. Define $\Phi(Co(F))$, the potential of $Co(F)$, to be

$$\Phi(Co(F)) = \sum_{T \in Co(F)} ht(T) = \sum_{T \in Ta(F)} ht(T) + |St(F)|.$$

The value of $\Phi(Co(F))$ is well-defined since $Ta(F)$ is the set of all tall trees of F and $St(F)$ is a set of minimum cardinality.

For $T \in Co(F)$, we define $\Phi(Cl(T), T)$, the potential of cluster $Cl(T)$ with core T, by considering four cases. In essence, $\Phi(Cl(T), T)$ is the difference between the height of T and the height of the tallest tree that can be formed by merging only stars of $Cl(T)$ with T.

Case 1: $Cl(T)$ is empty. In this case, define $\Phi(Cl(T), T) = 0$.

Case 2: If the *id* of the root of T is smaller than the *id* of the root of every star of $Cl(T)$, then the core of the cluster is *grounded*. If T is grounded, then no stars of $Cl(T)$ can adopt T.

But any number of stars of $Cl(T)$ may be adopted by T. Since every star of $Cl(T)$ is adjacent to T, regardless of the number of stars that T adopts, the height of the tallest tree that can be formed is $ht(T)+1$. In this case, define $\Phi(Cl(T),T)=1$.

Case 3: If T is not grounded and $Cl(T)$ contains only one star S, then S may adopt T. The height of the tallest tree that can be formed is $ht(T)+1$. In this case, define $\Phi(Cl(T),T)=1$.

Case 4: If T is not grounded and $Cl(T)$ contains at least two stars, then at least one star of $Cl(T)$ has a root whose *id* is smaller than the *id* of the root of T. During the first phase, at most one star of $Cl(T)$ may adopt T, and T may adopt any number of stars of $Cl(T)$. The height of the tallest tree T' that can be formed is $ht(T)+2$. The algorithm performs at least two shortcut operations before the hooking operation in the second phase. Thus the height of T' is reduced so that

$$ht(T') \le \lceil \lceil (ht(T)+2)/2 \rceil /2 \rceil = \lceil (ht(T)+2)/4 \rceil .$$

Since $ht(T') \le ht(T)$, the height of the tallest tree that can be formed through hooking operations in the second phase is at most $ht(T)+2$. In this case, define $\Phi(Cl(T),T)=2$. This concludes the last case.

Let R be the sum of the potentials of the clusters whose cores are trees of $Co(F)$, i.e.,

$$R = \sum_{T \in Co(F)} \Phi(Cl(T),T).$$

Different assignments of stars of $Cl(F)$ to trees of $Co(F)$ yield different clusters $Cl(T)$. Since the potential of a cluster is determined by the number of stars in the cluster, the *id* of the root of the core of the cluster, and the *id*'s of the roots of the stars in the cluster, different assignments of stars of $Cl(F)$ to trees of $Co(F)$ may yield different values for R. Define

$\Phi(Cl(F))$, the potential of $Cl(F)$, to be the maximum value of R over all possible sets $St(F)$ and assignments of stars of $Cl(F)$ to trees of $Co(F)$. Note that $\Phi(Cl(F)) \leq |Cl(F)|$ since by definition $\Phi(Cl(T),T) \leq |Cl(T)|$. Finally, define $\Phi(F)$, the potential of the forest F, to be

$$\Phi(F) = \Phi(Co(F)) + \Phi(Cl(F)).$$

4.6.2.2. Contribution

Suppose that the algorithm is at the beginning of phase k. For a particular connected component C, let F_k be the forest of trees in PG that contains the vertices of $V(C)$. During phases k and $k+1$ of the algorithm, only some of the trees of F_k may be merged. Some tall trees may remain stagnant but be reduced in height through shortcut operations, and some stars may remain stagnant and unchanged. Since the potential of F_k is based on clusters, we need to account for the change in potential when only some of the stars in a cluster are merged. We define a measure which we call *contribution* to relate the potential of F_{k+2} to the potential of F_k.

Let forest F_{k+2} be the set of trees of PG that contain the vertices of $V(C)$ at the end of phase $k+1$. Then each tree of F_{k+2} contains the vertices of one or more trees of F_k. If a tree T' of F_{k+2} contains all of the vertices of a tree T of F_k, then we say that T' *incorporates* T.

If T' is a tree of F_{k+2} that is not in F_k, then T' is a tree formed as a result of shortcut operations on either a tall tree of F_k or a tree that was created by merging two or more trees of F_k. Let H be a subset of trees of F_k incorporated into T'. The *contribution* of H to T', written $cont(H,T')$, which we define precisely below, is a rough measure of the portion of the height of T' that is due to trees of H.

Let l be the length of the longest path in T' consisting only of vertices of trees of H. If $l \geq 1$, then $cont(H,T') = l$. If $l = 0$, i.e., no two vertices of trees of H are joined by an arc of PG, then $cont(H,T') = 1$. Note that if T' incorporates only trees of H, then $cont(H,T') = ht(T')$.

Let $Stag(T)$ be the set of stars of $Cl(T)$ that were stagnant during phases k and $k+1$. Note that $Stag(T)$ could be empty. Let T' be the tree of F_{k+2} that incorporates T. The contribution of $Stag(T)$ to T' is the potential $Stag(T)$ would have if T' were the core of a cluster that comprised only the stars of $Stag(T)$.

We clarify the concept of contribution with an example. Suppose that at the beginning of phase k the forest F_k contains the tall tree and its cluster shown in Figure 4.2. Arrows indicate arcs of PG and dotted lines indicate edges of G. The numbers are the *id*'s of the

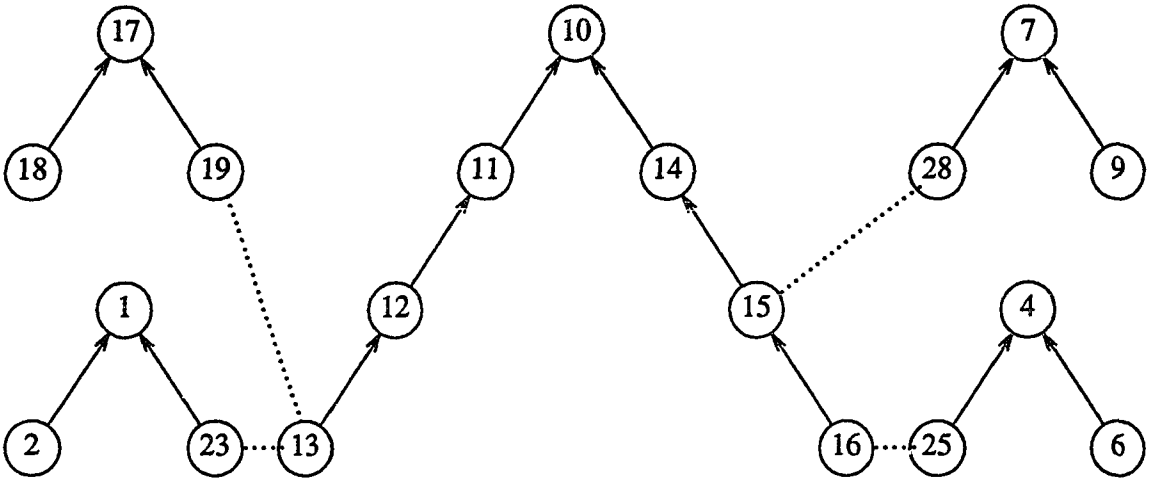

Figure 4.2. An example of a cluster.

vertices. We will refer to trees by the *id*'s of their roots. In the example, tree T_{10} is the core of a tall tree cluster and $Cl(T_{10}) = \{T_1, T_4, T_7, T_{17}\}$. Since T_{10} is the core of a cluster, $\Phi(T_{10}) = 3$. Since T_{10} is not grounded and $Cl(T_{10})$ contains at least two stars, $\Phi(Cl(T_{10}), T_{10}) = 2$.

Suppose that during Part 1 of phase k the algorithm performs one shortcut operation on each vertex of the tree T_{10}. Then at the end of Part 1 of phase k, the height of T_{10} is 2. During Part 2 of phase k, suppose that vertex 7 adopts T_{10} and vertex 11 of tree T_{10} adopts T_{17}, while T_1 and T_4 remain stagnant. The height of the new tree T' that is formed is 4. After the algorithm performs at least one shortcut operation in each of Part 3 of phase k and Part 1 of phase $k+1$, the height of T' is reduced to 1. During Part 2 of phase $k+1$, suppose T' is adopted by vertex 1 to form a new tree T'' while again T_4 remains stagnant. After a shortcut operation, the height of T'' is 1. Thus, at the end of phase $k+1$, the contribution of the set of trees $\{T_1, T_7, T_{10}, T_{17}\}$ to T'' is 1. If T'' were the core of a cluster containing only the star T_4, then the potential of the cluster would be 1. Thus, $cont(\{T_4\}, T'') = 1$.

4.6.2.3. Performance

To analyze the progress of the algorithm, we examine one execution of the algorithm. But the analysis holds for every execution. Assume that at the beginning of phase k we know a priori how the trees of F_k will be merged and the stars of F_k that will remain stagnant during the next two phases. Let $St(F_k)$ be the set of stars of F_k and let A_k^* denote the assignment of stars of $Cl(F_k)$ to trees of $Co(F_k)$ that determines $\Phi(Cl(F_k))$. For $St(F_k)$, we define another assignment A_k, where the stars of $Cl(F_k)$ are assigned to trees of $Co(F_k)$ in the following order.

Let S be any star of $Cl(F_k)$. If S is adopted by a vertex of a tree T in $Co(F_k)$ during phase k or $k+1$, then assign S to T. If a vertex of S adopts a tree T of $Co(F_k)$ during phase k or $k+1$ and S has not already been assigned to some other tree, then assign S to T. If S is stagnant during phases k and $k+1$ and S is a star of $Cl(F_{k+2})$, then let T' be the tree of $Co(F_{k+2})$ to which S is assigned in A^*_{k+2}. In A_k, assign S to a tree T adjacent to S in F_k that is incorporated into T'. Finally, for the set of stagnant stars of $Cl(F_k)$ that are stars of $Co(F_{k+2})$, find an assignment of those stars to trees of $Co(F_k)$ such that the sum of their contributions to trees of F_{k+2} is maximized. This completes the specification of A_k.

We now define $Cl(T) = \{S \in Cl(F_k) : A_k(S) = T\}$. For the assignment A_k, define $\Psi(Co(F_k))$ to be

$$\Psi(Co(F_k)) = \sum_{T \in Co(F_k)} ht(T),$$

define $\Psi(Cl(F_k))$ to be

$$\Psi(Cl(F_k)) = \sum_{T \in Co(F_k)} \Phi(Cl(T), T),$$

and define $\Psi(F_k)$ to be

$$\Psi(F_k) = \Psi(Co(F_k)) + \Psi(Cl(F_k)).$$

For convenience, let ϕ_k be the value of $\Phi(F_k)$, and let ψ_k be the value of $\Psi(F_k)$. By the definition of $\Phi(F_k)$, it follows that $\psi_k \leq \phi_k$.

Lemma 4.6.6: If every tree of F_k participates in at least one adoption during phases k and $k+1$, then $\sum_{T' \in F_{k+2}} ht(T') \leq \phi_k$.

Proof: Let T' be a tree of F_{k+2}. Let M be the set of trees of F_k that are merged to form T'. By Lemma 4.6.5 and from the assignment A_k, M can be partitioned into subsets where each subset contains exactly one tree T of $Co(F_k)$ and the set of stars $Cl(T)$ that are merged with T. A set of trees $\{T_1, T_2, \ldots, T_k\}$ is *merged in series* if T_1 adopts T_2, T_2 adopts T_3, ..., and T_{k-1} adopts T_k. The height of T' is greatest if all of the trees of M are merged in series.

If T is a tree of $Co(F_k)$, then $\Phi(T) + \Phi(Cl(T), T)$ is an upper bound on the height of the tree that can be formed by merging T and the stars of $Cl(T)$ in series. Let $Co(M)$ be the set of trees of $Co(F_k)$ in M. Then

$$ht(T') \leq \sum_{T \in Co(M)} (\Phi(T) + \Phi(Cl(T), T))$$

because the sum of the potentials of the cores and their corresponding clusters is an upper bound on the height of the tree that can be formed by merging all trees of M in series. Since every tree of F_k participates in at least one adoption during phases k and $k+1$, every tree of F_k is incorporated into some tree of F_{k+2}. Since T' may be any tree of F_{k+2}, it follows from the assignment A_k and the definition of $\Psi(F_k)$ that

$$\sum_{T' \in F_{k+2}} ht(T') \leq \psi_k.$$

Then, by the definition of potential, $\psi_k \leq \phi_k$, and the result follows. □

In general, the sum of the heights of the trees that are formed will be less than ϕ_k. The trees that are formed depend on the relative ordering of the *id*'s of the roots of the trees and the order in which trees are merged. Thus, the actual number of trees that are merged in series may be smaller than the number used in the determination of the potential. Also, the root of a tree may be adopted by a vertex that is closer to the root than a parent of a leaf

vertex. Finally, a star of $Cl(F_k)$ that is adjacent to several trees of $Co(F_k)$ might not be merged with the tree to which it was assigned when determining the potential.

Lemma 4.6.7: The sum of the contributions of the trees of F_k to the trees of F_{k+2} is at most (3/5) ϕ_k.

Proof: Since $\psi_k \leq \phi_k$, it suffices to show that after two phases, the sum of the contributions of the trees of F_k to the trees of F_{k+2} is at most (3/5) ψ_k. We will partition the trees of F_k into subsets such that for each subset, the sum of the contributions of the trees in the subset to a tree of F_{k+2} is at most 3/5 the sum of the potentials of those trees in F_k.

Let T be a tree of $Co(F_k)$ and let r be the root of T. During phases k and $k+1$, stars of $Cl(T)$ may be merged with T. Let $M'(T)$ be the set of stars of $Cl(T)$ that are merged with T during phase k. At the end of phase k, let T' be the tree of F_{k+1} that incorporates T. Let $M''(T)$ be the set of stars of $Cl(T)$ that are stagnant during phase k and merged with T' during phase $k+1$. At the end of phase $k+1$, let T'' be the tree of F_{k+2} that incorporates T. Finally, let $M(T) = M'(T) \cup M''(T)$.

Let $Cl(T) + T$ denote the set of trees $Cl(T) \cup \{T\}$. Let $Stag(T)$ denote the set of stars of $Cl(T)$ that are stagnant during phases k and $k+1$. Note that $Cl(T) = M(T) \cup Stag(T)$. Define ρ to be

$$\rho = \frac{cont(M(T)+T, T'') + cont(Stag(T), T'')}{\Phi(T) + \Phi(Cl(T), T)}$$

$$= \frac{cont(M(T)+T, T'') + cont(Stag(T), T'')}{ht(T) + \Phi(Cl(T), T)}.$$

We consider seven cases. In each case, we show that $\rho \leq 3/5$.

Case 1: T is a star, T is grounded, and $Cl(T)$ contains at least one star. Then $ht(T) + \Phi(Cl(T), T) = 2$. Since T is grounded, by Lemma 4.6.1, at the end of the hooking

operation in phase k, every star of $Cl(T)$ is adopted by T; hence $M'(T) = Cl(T)$. After at least one shortcut operation, at the end of phase k, every vertex of $Cl(T)+T$ has a parent whose *id* is at most the *id* of r. Since every vertex of $Cl(T)+T$ has a parent that is either r or an ancestor of r in T', the length of the longest path in T' consisting only of vertices of the trees of $Cl(T)+T$ is at most 1. Therefore, by the definition of contribution, $cont(Cl(T)+T, T') = 1$. After further shortcut operations, at the end phase k+1, each vertex of $Cl(T)+T$ has a parent whose *id* is at most the *id* of the parent of the vertex at the end of phase k. Thus, $cont(Cl(T)+T, T'') = 1$. Since no stars of $Cl(T)$ are stagnant, $Stag(T)$ is empty. Thus, $cont(Stag(T), T'') = 0$. It follows that $\rho = 1/2$.

Case 2: T is a star, T is not grounded, and $Cl(T)$ contains at least one star. Then $ht(T) + \Phi(Cl(T), T) \geq 2$. Let $min\text{-}id(Cl(T))$ denote the *id* of the root of the star in $Cl(T)$ whose *id* is smallest. Then, at the end of the hooking operation in phase k, the parent of r is a vertex b, where $b \leq min\text{-}id(Cl(T))$. After a shortcut operation, at the end of phase k, every vertex of T has a parent whose *id* is at most b, and every vertex of a star of $M'(T)$ has a parent whose *id* is at most r.

In phase k+1, after at least one shortcut operation, every vertex of a star of $M'(T)$ has a parent whose *id* is at most b. Since every vertex of T has a parent whose *id* is at most b and $b \leq min\text{-}id(Cl(T))$, by Lemma 4.6.1, at the end of the hooking operation in phase k+1, every star of $Cl(T)$ that is stagnant during phase k is adopted by T'; hence, $M(T) = Cl(T)$. After a shortcut operation, every vertex of a star of $M''(T)$ has a parent whose *id* is at most b. Thus, at the end of phase k+1, every vertex of a tree of $Cl(T)+T$ has a parent whose *id* is at most b. Since $b \leq min\text{-}id(Cl(T))$, it follows, as in Case 1, that $cont(Cl(T)+T, T'') = 1$. Since $Stag(T)$ is empty, $cont(Stag(T), T'') = 0$. Thus, $\rho \leq 1/2$.

Case 3: The height of T is 2 and $Cl(T)$ contains at least one star. Then $ht(T) + \Phi(Cl(T), T) \geq 3$. After the first shortcut operation during phase k, the remainder of the analysis for Case 3 is the same as the analysis for Case 1 or Case 2, depending on whether T is grounded. In either case, $cont(Cl(T) + T, T'') = 1$, and thus $\rho \leq 1/3$.

Case 4: T is a tall tree and $Cl(T)$ contains no stars. Since T is a tall tree, $\Phi(T) = ht(T) \geq 2$. Since $Cl(T)$ is empty, $\Phi(Cl(T), T) = 0$. After at least two shortcut operations during each of phases k and $k+1$, at the end of phase $k+1$,

$$cont(T, T'') \leq \lceil \lceil \lceil \lceil (ht(T)/2) \rceil /2 \rceil /2 \rceil /2 \rceil \qquad (4.1)$$
$$= \lceil ht(T)/16 \rceil .$$

Because $Cl(T)$ is empty, $Stag(T)$ is empty, and $cont(Stag(T), T'') = 0$. If $2 \leq ht(T) \leq 16$, then by Inequality (4.1), $cont(T, T'') = 1$. Since $ht(T) \geq 2$, it follows that $\rho \leq 1/2$.

Eliminating the ceiling operations in Inequality (4.1), we obtain $cont(T, T'') \leq ht(T)/16 + 1$. If $ht(T) \geq 17$, then it follows that $\rho \leq 33/272 < 3/5$.

Case 5: The height of T is at least 3, T is grounded, and $Cl(T)$ contains at least one star. Then $\Phi(T) + \Phi(Cl(T), T) = ht(T) + 1$. After at least one shortcut operation followed by a hooking operation in which T is grounded and at least one more shortcut operation, at the end of phase k,

$$cont(M'(T) + T, T') \leq \lceil (\lceil ht(T)/2 \rceil + 1)/2 \rceil .$$

Since T is incorporated into T', the *id* of the root of T' is at most the *id* of the root of T. Thus, T' is grounded with respect to the stars of $Cl(T)$. After at least one more shortcut

operation followed by another hooking operation in which T' is grounded and then at least one more shortcut operation, at the end of phase $k+1$,

$$cont(M(T)+T,T'') \le \lceil (\lceil \lceil (\lceil ht(T)/2 \rceil + 1)/2 \rceil /2 \rceil + 1)/2 \rceil \qquad (4.2)$$
$$= \lceil (\lceil (\lceil ht(T)/2 \rceil + 1)/4 \rceil + 1)/2 \rceil .$$

In phases k and $k+1$, the algorithm performs at least three shortcut operations before the hooking operation in phase $k+1$. If $ht(T) \le 8$, then before the hooking operation in phase $k+2$, every vertex of T has a parent whose *id* is at most the *id* of r. Since T is grounded, by Lemma 4.6.1, every star of $Cl(T)$ is adopted. Thus, $Stag(T)$ is empty, and $cont(Stag(T),T'') = 0$.

If $3 \le ht(T) \le 8$, then by Inequality (4.2), $cont(Cl(T)+T,T'') \le 2$. Since $ht(T) + \Phi(Cl(T),T) \ge 4$, $\rho \le 1/2$.

If $ht(T) \ge 9$, then stars of $Cl(T)$ may remain stagnant. Since T'' incorporates T, the *id* of the root of T'' is at most the *id* of r. Thus T'' is grounded with respect to the stars of $Stag(T)$. By the definition of contribution, $cont(Stag(T),T'') \le 1$. Eliminating the ceiling operations in Inequality (4.2), we obtain $cont(M(T)+T,T'') \le ht(T)/16 + 9/4$. Since $ht(T) \ge 9$, it follows that $\rho \le 61/160 < 3/5$.

Case 6: The height of T is at least 3, T is not grounded, and $Cl(T)$ contains at least one star. We first consider the case where $Cl(T)$ has only one star.

If $Cl(T)$ contains one star S, then $\Phi(T) + \Phi(Cl(T),T) = ht(T) + 1$. If S and T are incorporated into a tree T'' of F_{k+2}, then the value of $cont(S+T,T'')$ is greatest when S adopts T in phase $k+1$, because then the algorithm performs fewer shortcut operations on the tree that is formed. Thus,

$$cont(M(T)+T,T'') \le \lceil (\lceil \lceil \lceil ht(T)/2 \rceil /2 \rceil /2 \rceil + 1)/2 \rceil \qquad (4.3)$$
$$= \lceil (\lceil ht(T)/8 \rceil + 1)/2 \rceil .$$

If $3 \le ht(T) \le 8$, then by Inequality (4.3), $cont(M(T)+T,T'') = 1$. Since $Cl(T)$ contains only one star, $cont(Stag(T),T'') \le 1$. Since $ht(T) \ge 3$ and $\Phi(Cl(T),T) = 1$, it follows that $\rho \le 1/2$.

Eliminating the ceiling operations in Inequality (4.3), we obtain $cont(M(T)+T,T'') \le ht(T)/16 + 2$. Since $cont(Stag(T),T'') \le 1$ and $\Phi(Cl(T),T) = 1$, it follows that if $ht(T) \ge 9$, then $\rho \le 57/160 < 3/5$.

If $Cl(T)$ contains at least two stars, then $\Phi(T) + \Phi(Cl(T),T) = ht(T) + 2$. After at least one shortcut operation followed by a hooking operation in which T is not grounded and at least one more shortcut operation, at the end of phase k,

$$cont(M'(T)+T,T') \le \lceil (\lceil ht(T)/2 \rceil + 2)/2 \rceil .$$

After at least one more shortcut operation followed by another hooking operation in which T' may not be grounded and then at least one more shortcut operation, at the end of phase $k+1$,

$$cont(M(T)+T,T'') \le \lceil (\lceil \lceil (\lceil ht(T)/2 \rceil + 2)/2 \rceil /2 \rceil + 2)/2 \rceil \qquad (4.4)$$
$$= \lceil (\lceil (\lceil ht(T)/2 \rceil + 2)/4 \rceil + 2)/2 \rceil .$$

If $3 \le ht(T) \le 4$, then at the end of phase k, after at least two shortcut operations, every vertex of T has a parent whose *id* is at most the *id* of r. After the hooking operation in phase $k+1$, the parent of r is a vertex b, where $b \le min\text{-}id(Cl(T))$. After at least one more shortcut operation, at the end of phase $k+1$, every vertex of T has a parent whose *id* is at most b. Vertex b cannot be the root of a star in $Stag(T)$ because either $b < min\text{-}id(Cl(T))$ or b

is the root of a star that participated in an adoption. Since the *id* of the root of every star of $Stag(T)$ is larger than b, T'' is grounded with respect to the stars of $Stag(T)$. Thus $cont(Stag(T), T'') \leq 1$. By Inequality (4.4), $cont(M(T)+T, T'') \leq 2$. Since $ht(T) \div \Phi(Cl(T), T) \geq 5$, it follows that $\rho \leq 3/5$.

If $5 \leq ht(T) \leq 12$, then by Inequality (4.4), $cont(M(T)+T, T'') \leq 2$. Since $cont(Stag(T), T'') = \Phi(Stag(T), T'') \leq 2$ and $\Phi(Cl(T), T) = 2$, if $ht(T) \geq 5$, then $\rho \leq 4/7 < 3/5$.

Eliminating the ceiling operations in Inequality (4.4), we obtain $cont(M(T)+T, T'') \leq ht(T)/16 + 23/8$. Since $cont(Stag(T), T'') \leq 2$ and $\Phi(Cl(T), T) = 2$, it follows that if $ht(T) \geq 13$, then $\rho \leq 91/240 < 3/5$.

Case 7: T is a star and $Cl(T)$ contains no stars. Since no stars are assigned to T, we call T a *free* star. Since T is a star of $St(F_k)$, there is at least one internal star adjacent to T. By Corollary 4.6.3, T participates in at least one adoption during phases k and $k+1$.

Since T is the core of a cluster, $\Phi(T) = 1$. If T is merged only with other free stars, then the sum of the potentials of the free stars is the number of stars. Let Q be the set of free stars of F_k including T that are merged during phases k and $k+1$. We can partition the stars of Q into subsets such that for each subset, one star S serves as the core of a cluster $Cl(S)$ and the remaining stars in the subset are the stars of $Cl(S)$. Using the analysis for Cases 1 and 2, we see that the contribution of the stars in each group to the tree T'' of F_{k+2} that incorporates them is at most half the sum of their potential.

If T is not merged with another free star, then suppose we consider T alone. At the end of phase $k+1$, $cont(T, T'') = 1$. In this case, $\rho = 1$. But we want $\rho \leq 3/5$. Let H be a set of trees of $Cl(T)+T$ from one of the Cases 1 through 6 above. We show that if T is added to

H, then for the augmented set of trees $H+T$, the sum of the contributions of the trees of $H+T$ to a tree of F_{k+2} is at most 3/5 the sum of the potentials of those trees in F_k. We consider four subcases.

Subcase 7.1: T adopts a vertex v that is the root of a tree T_c adjacent to T. Since T is free, T adopts no stars. Otherwise at least one star would have been assigned to T. Thus T_c is a tall tree and the core of a cluster. We apply the analysis for Cases 3 and 6 in which T_c is the core of a cluster containing the stars $Cl(T_c)+T$ and T_c is not grounded.

Subcase 7.2: T is adopted by a tree T_c adjacent to T that is the core of a cluster. We apply the analysis for the case in which T_c is the core of a cluster containing the stars $Cl(T_c)+T$. The addition of T has no effect on whether T_c is grounded.

Subcase 7.3: T is adopted by a star S in phase k. Let T_c be the tree for which $S \in Cl(T_c)$. Since S is assigned to T_c, in phase k either S adopts T_c or T_c adopts S. We consider the two subcases.

Subcase 7.3.1: S adopts T_c. Since T is a star, $ht(T) \leq ht(T_c)$. It follows from the analysis of the contribution in Cases 1 through 6 above that $cont((M(T_c)+T_c+T),T'') = cont(M(T_c)+T_c,T'')$. Thus we can add T to the set of trees $Cl(T_c)+T_c$ and the analysis remains valid.

Subcase 7.3.2: T_c adopts S. Consider the sum of the contributions of the set of trees $Cl(T_c)+T_c+T$ to a tree T'' of F_{k+2}. Define ρ' to be

$$\rho' = \frac{cont(M(T_c)+T_c+T,T'')+cont(Stag(T_c),T'')}{\Phi(T_c)+\Phi(Cl(T_c),T_c)+\Phi(T)}$$

$$= \frac{cont(M(T_c)+T_c+T,T'')+cont(Stag(T_c),T'')}{ht(T_c)+\Phi(Cl(T_c),T_c)+1}.$$

We show that $\rho' \leq 1/2$. Let r_c be the root of T_c. First suppose that T_c is grounded. Since T_c adopts S and S adopts T in phase k, trees T_c, S, and T are merged in series. After at least one shortcut operation followed by a hooking operation in which T_c, S, and T are merged in series and at least one more shortcut operation, at the end of phase k,

$$cont(M'(T_c) + T_c + T, T') \leq \left\lceil (\left\lceil ht(T_c)/2 \right\rceil + 2)/2 \right\rceil .$$

After at least one more shortcut operation followed by another hooking operation in which T' is grounded and then at least one more shortcut operation, at the end of phase $k+1$,

$$\begin{aligned} cont(M(T_c) + T_c + T, T'') &\leq \left\lceil (\left\lceil \left\lceil (\left\lceil ht(T_c)/2 \right\rceil + 2)/2 \right\rceil /2 \right\rceil + 1)/2 \right\rceil \\ &= \left\lceil (\left\lceil (\left\lceil ht(T_c)/2 \right\rceil + 2)/4 \right\rceil + 1)/2 \right\rceil . \end{aligned} \tag{4.5}$$

If $ht(T_c) \leq 8$, then by the analysis for Cases 1, 3, and 5, at the end of phase $k+1$, every star of $Cl(T_c)$ is adopted, and thus $cont(Stag(T_c), T'') = 0$.

If $ht(T_c) \leq 4$, then by Inequality (4.5), $cont(M(T_c) + T_c + T, T'') = 1$. Since $ht(T_c) \geq 1$, $\Phi(Cl(T_c), T_c) = 1$, and $ht(T) = 1$, it follows that $\rho' \leq 1/3$.

If $5 \leq ht(T_c) \leq 8$, then by Inequality (4.5), $cont(M(T_c) + T_c + T, T'') \leq 2$. Since $ht(T_c) \geq 5$, $\Phi(Cl(T_c), T_c) = 1$, and $ht(T) = 1$, it follows that $\rho' \leq 2/7 < 1/2$.

Eliminating the ceiling operations in Inequality (4.5), we obtain $cont(M(T_c) + T_c + T, T'') \leq ht(T_c)/16 + 19/8$. Since $cont(Stag(T_c), T'') \leq 1$, $\Phi(Cl(T_c), T_c) = 1$, and $ht(T) = 1$, it follows that if $ht(T_c) \geq 9$, then $\rho' \leq 63/176 < 1/2$.

Now suppose that T_c is not grounded. If $1 \leq ht(T_c) \leq 2$, then by the analysis for Cases 2 and 3, after the hooking operation in phase k, the parent of r_c is a vertex b, where $b \leq min\text{-}id(Cl(T_c))$. By Lemma 4.6.1, at the end of phase $k+1$, every star of $Cl(T_c)$ is

adopted. Thus, $M(T_c) = Cl(T_c)$ and $cont(Stag(T_c), T'') = 0$. Since T_c adopts S and S adopts T during the hooking operation in phase k, after at least three shortcut operations, at the end of phase $k+1$, every vertex of T has a parent whose *id* is at most b. Thus, as in the analysis for Case 1, it follows that $cont(Cl(T_c) + T_c + T, T'') = 1$. Since $ht(T_c) \geq 1$, $\Phi(Cl(T_c), T_c) \geq 1$, and $ht(T) = 1$, it follows that $\rho' \leq 1/3$.

In phase k, trees T_c, S, and T are merged in series. Since T_c adopts S and T_c is not grounded, $Cl(T_c)$ contains at least two stars. After at least one shortcut operation followed by a hooking operation in which T_c is not grounded and at least one more shortcut operation, at the end of phase k,

$$cont(M'(T_c) + T_c + T, T') \leq \left\lceil (\left\lceil ht(T_c)/2 \right\rceil + 3)/2 \right\rceil .$$

After at least one more shortcut operation followed by another hooking operation in which T' may not be grounded and then at least one more shortcut operation, at the end of phase $k+1$,

$$\begin{aligned} cont(M(T_c) + T_c + T, T'') &\leq \left\lceil (\left\lceil \left\lceil (\left\lceil ht(T_c)/2 \right\rceil + 3)/2 \right\rceil /2 \right\rceil + 2)/2 \right\rceil \\ &= \left\lceil (\left\lceil (\left\lceil ht(T_c)/2 \right\rceil + 3)/4 \right\rceil + 2)/2 \right\rceil . \end{aligned} \tag{4.6}$$

If $3 \leq ht(T_c) \leq 4$, then by the analysis of Case 6, $cont(Stag(T_c), T'') \leq 1$. By Inequality (4.6), $cont(M(T_c) + T_c + T, T'') \leq 2$. Since $ht(T_c) \geq 3$, $\Phi(Cl(T_c), T_c) = 2$, and $ht(T) = 1$, it follows that $\rho' \leq 1/2$.

If $5 \leq ht(T_c) \leq 10$, then by Inequality (4.6) $cont(M(T_c) + T_c + T, T'') \leq 2$. By the definition of contribution, $cont(Stag(T_c), T'') \leq 2$. Since $ht(T_c) \geq 5$, $\Phi(Cl(T_c), T_c) = 2$, and $ht(T) = 1$, it follows that $\rho' \leq 1/2$.

Finally, eliminating the ceiling operations in Inequality (4.6), we obtain $cont(M(T_c) + T_c + T, T'') \leq ht(T_c)/16 + 3$. Since $cont(Stag(T_c), T'') \leq 2$, $\Phi(Cl(T_c), T_c) = 2$, and $ht(T) = 1$, it follows that if $ht(T_c) \geq 11$, then $\rho' \leq 13/32 < 1/2$.

Subcase 7.4: T remains stagnant during phase k, and in phase $k+1$ tree T is adopted by a vertex v of a tree T_v not adjacent to T in F_k. Let T_c be the tree of $Co(F_k)$ for which T_v is a tree of $Cl(T_c) + T_c$. We show that T can be added to the set of trees $Cl(T_c) + T_c$ without increasing ρ.

Since T_v is not adjacent to T, some tree T' of F_{k+1} incorporates T_v and a tree adjacent to T in F_k. Otherwise, in phase $k+1$, vertex v and the root of T would not form an eligible pair.

Since v is a vertex of T_v, and thus a vertex of T', in phase $k+1$, tree T' adopts T. In the analysis for Cases 2 through 6, the determination of the contribution is valid for an arbitrary number of stars adopted by T'. Thus, by adding T to the set of trees $Cl(T_c) + T_c$, the analysis still holds.

For Case 1, at the end of phase k, every star of $Cl(T_c)$ is adopted by T_c. Thus, $cont(Cl(T_c) + T_c, T') = 1$. In phase $k+1$, during the hooking operation T' adopts T. After at least one shortcut operation, every vertex of T has a parent whose *id* is at most the *id* of the root of T_c. Thus, $cont(Cl(T_c) + T_c + T, T'') = 1$. By the analysis from Case 1, $\rho = 1/2$. This completes the last subcase.

From these four subcases, we conclude that if T is added to some set H from one of the Cases 1 to 6 above, then for the set of trees $H + T$, $\rho \leq 3/5$. This completes Case 7.

Since every tree of F_k is included in one of the above cases, the sum of the contributions of the trees of F_k to trees of F_{k+2} is at most (3/5) ψ_k. Since $\psi_k \leq \phi_k$, the result follows. □

We give an example to illustrate the computation of ρ. Consider the cluster of Figure 4.2. Suppose in F_k tree T_{10} is the core of a cluster containing four stars and T_{10} is not grounded. Thus, $\Phi(T_{10}) + \Phi(Cl(T_{10}), T_{10}) = 3 + 2 = 5$.

In phase k, suppose every processor performs one shortcut operation before the hooking operation. After the shortcut operation, the height of T_{10} is reduced to 2. During the hooking operation, processor $p(15,28)$ hooks vertex 10 to vertex 7, and processor $p(13,19)$ hooks vertex 17 to vertex 11. Thus tree T_7 adopts tree T_{10} and tree T_{10} adopts tree T_{17}. Tree T' incorporates trees T_7, T_{10}, and T_{17}.

After at least two shortcut operations, before the hooking operation in phase k+1, the parent of every vertex of T' is vertex 7. During the hooking operation in phase k+1, tree T' is adopted by tree T_1 to form a new tree T''. Suppose tree T_4 remains stagnant. After at least one shortcut operation, at the end of phase k+1, the height of T'' is 1. Thus $cont(\{T_1,T_7,T_{10},T_{17}\},T'') = 1$ and $cont(\{T_4\},T'') = 1$. It follows then that $\rho = 2/5$. This concludes the example.

A tree of F_k is *active* if the tree participated in an adoption or was reduced in height by a shortcut operation during phases k and k+1. If T'' is a tree of F_{k+2} that is not in F_k, then T'' incorporates one or more active trees of F_k. Call tree T'' a *marked* tree. Let $Stag(F_k)$ be the stars of F_k that remain stagnant during phases k and k+1. Call a stagnant star of $Stag(F_k)$ an *unmarked* star.

Let $Ma(F_{k+2})$ be the set of marked trees of F_{k+2}. Each tree T'' of $Ma(F_{k+2})$ incorporates trees from one or more of the subsets of trees considered in the various cases in the proof of Lemma 4.6.7. Let $In(Co(F_k),T'')$ be the set of trees of $Co(F_k)$ incorporated into T''. For a tree T of $In(Co(F_k),T'')$, let $cont(M(T)+T,T'')$ be the contribution of the

set of trees comprising T and the stars of $Cl(T)$ merged with T to T''. By the definition of contribution,

$$\sum_{T \in In(Co(F_k), T'')} cont(M(T) + T, T'') \geq ht(T''). \tag{4.7}$$

Define

$$\sigma_{k+2} = \sum_{T'' \in Ma(F_{k+2})} \left(\sum_{T \in In(Co(F_k), T'')} (cont(M(T) + T, T'') + cont(Stag(T), T'')) \right).$$

The value of σ_{k+2} is the sum of the contributions of the trees of F_k to the marked trees of F_{k+2}. The unmarked trees of F_{k+2} are stagnant stars of F_k, and thus their contribution to trees of F_{k+2} is included in σ_{k+2}.

Lemma 4.6.8: $\Phi(F_{k+2}) \leq \sigma_{k+2}$.

Proof: Every tall tree of F_{k+2} is formed as a result of a shortcut operation on a tall tree of F_k or a shortcut operation on a tall tree created by merging trees of F_k and is thus marked. If S is a star of $Stag(F_k)$, then by Corollary 4.6.4, every tree in F_k adjacent to S is a tall tree. Thus, every tall tree of F_k adjacent to S is incorporated into a marked tree in F_{k+2}. It follows then that every tree adjacent to S in F_{k+2} is a marked tree. Recall that ϕ_{k+2} is the value of $\Phi(F_{k+2})$. We now show that $\phi_{k+2} \leq \sigma_{k+2}$.

We first consider the case in which every star of $St(F_{k+2})$ is marked. Then every tree of $Co(F_{k+2})$ is marked because every tree of $Ta(F_{k+2})$ is marked. Since every tree of $Co(F_{k+2})$ is marked, we can directly compare σ_{k+2} with $\Phi(F_{k+2})$.

By the definition of the potential function,

$$\Phi(Co(F_{k+2})) = \sum_{T'' \in Co(F_{k+2})} ht(T''). \tag{4.8}$$

It follows from Inequality (4.7) and Equation (4.8) that for the trees T'' that are the cores of clusters in F_{k+2},

$$\sum_{T'' \in Co(F_{k+2})} \Big(\sum_{T \in In(Co(F_k), T'')} cont(M(T) + T, T'') \Big) \geq \Phi(Co(F_{k+2})). \tag{4.9}$$

By the definition of the potential function, $\Phi(Cl(F_{k+2}))$ is determined by a set $St(F_{k+2})$ of minimum cardinality and an assignment A^*_{k+2} of stars of $Cl(F_{k+2})$ to trees of $Co(F_{k+2})$ such that $\sum_{T'' \in Co(F_{k+2})} \Phi(Cl(T''), T'')$ is maximum. If S is an unmarked star, then since every star of $St(F_{k+2})$ is marked, S must be a star of $Cl(F_{k+2})$. Note that $Cl(F_{k+2})$ may also contain marked stars. Suppose we first assign only the unmarked stars of $Cl(F_{k+2})$ to the trees of $Co(F_{k+2})$ as given by A^*_{k+2}. Let T'' be a tree of $Co(F_{k+2})$. Then the stars of $Cl(T'')$ are stagnant stars of F_k. Specifically,

$$Cl(T'') = \bigcup_{T \in In(Co(F_k), T'')} Stag(T).$$

Let τ be the sum of the potentials of the clusters that are formed. Then

$$\tau = \sum_{T'' \in Co(F_{k+2})} \Phi(Cl(T''), T'').$$

By the definition of contribution, $cont(Stag(T), T'') = \Phi(Stag(T), T'')$. But since stagnant stars from more than one cluster of F_k may be contained in the same cluster of F_{k+2},

$$\sum_{T'' \in Co(F_{k+2})} \Big(\sum_{T \in In(Co(F_k), T'')} cont(Stag(T), T'') \Big) \geq \tau. \tag{4.10}$$

Now consider the marked stars of $Cl(F_{k+2})$. Let $Ma(Cl(F_{k+2}))$ be the set of marked stars of $Cl(F_{k+2})$. Let S be a star of $Ma(Cl(F_{k+2}))$. Suppose the assignment A^*_{k+2} assigns S to tree T'' of $Co(F_{k+2})$. Then the potential of the cluster containing the set of stars

$Cl(T'')+S$ is at most 1 greater than the potential of cluster $Cl(T'')$. After all stars of $Ma(Cl(F_{k+2}))$ are added to their assigned clusters, then by the definition of potential,

$$\Phi(Cl(F_{k+2})) \leq \tau + |Ma(Cl(F_{k+2}))|. \tag{4.11}$$

Since S is a marked star, by the definition of contribution,

$$\sum_{T \in In(Co(F_k),S)} cont(M(T)+T,S) \geq ht(S) = 1. \tag{4.12}$$

By Inequality (4.12), the sum of the contributions of the trees of F_k incorporated into each star S of $Ma(Cl(F_{k+2}))$ is at least 1. Thus, the sum of the contributions of trees of F_k to the stars of $Ma(Cl(F_{k+2}))$ is at least $|Ma(Cl(F_{k+2}))|$. It follows from Inequality (4.10) then that

$$\sum_{T'' \in Co(F_{k+2})} \Big(\sum_{T \in In(Co(F_k),T'')} cont(Stag(T),T'')) + \tag{4.13}$$

$$\sum_{S \in (Ma(Cl(F_{k+2})))} \Big(\sum_{T \in In(Co(F_k),S)} cont(M(T)+T,S)) \geq \tau + |Ma(Cl(F_{k+2}))|.$$

Finally, by Inequalities (4.7), (4.11), and (4.13), we conclude that $\sigma_{k+2} \geq \phi_{k+2}$.

Next we consider the case in which not every star of $St(F_{k+2})$ is marked. Let W be the set of stars of $Stag(F_k)$ that are stars of $St(F_{k+2})$. Let Y be the set of stars of $Cl(F_{k+2})$ covered by the stars of W. Since the stars of W were stagnant during phases k and $k+1$, by Corollary 4.6.3, no two stars of W are adjacent. By Corollary 4.6.4, every star of Y is marked. Also, $|Y| \geq |W|$. This is because if $|Y| < |W|$, then since the stars of Y cover the stars of W, $St(F_{k+2})$ would not be a set of minimum cardinality.

To show that $\sigma_{k+2} \geq \Phi(F_{k+2})$, we introduce a modified potential function $\Phi'(F_{k+2})$ similar to $\Phi(F_{k+2})$, except that the sets $St(F_{k+2})$ and $Cl(F_{k+2})$ are modified as follows. Every

star of W is moved from $St(F_{k+2})$ to $Cl(F_{k+2})$, and every star of Y is moved from $Cl(F_{k+2})$ to $St(F_{k+2})$. Then after this modification, in $\Phi'(F_{k+2})$, every star of $St(F_{k+2})$ is marked. From the analysis of the case in which every star of $St(F_{k+2})$ is marked, we conclude that $\sigma_{k+2} \geq \Phi'(F_{k+2})$.

Next, we show that $\Phi'(F_{k+2}) \geq \Phi(F_{k+2})$. In $\Phi(F_{k+2})$, for each star S of $St(F_{k+2})$, $\Phi(S) = 1$. Thus, the sum of the potentials of the stars of W contained in $St(F_{k+2})$ is $|W|$. If $Cl(S)$ is a set of stars contained in a cluster with core S, then the potential of the stars in $Cl(S)$ is at most $|Cl(S)|$. Thus, the sum of the potentials of the stars of Y contained in clusters of F_{k+2} is at most $|Y|$.

In $\Phi'(F_{k+2})$, since each star of Y is a star of $St(F_{k+2})$, the sum of the potentials of the stars of Y is $|Y|$. Since $|Y| \geq |W|$ and every star of W is adjacent to at least one star of Y, in $\Phi'(F_{k+2})$, each star of W may be assigned to a different star of Y. If each star of W belongs to a cluster containing a single star, then the sum of the potentials of the stars of W is $|W|$. It follows that $\Phi'(F_{k+2}) \geq \Phi(F_{k+2})$. From these two cases, we conclude that $\sigma_{k+2} \geq \Phi(F_{k+2})$. □

Theorem 4.6.9: Algorithm II requires at most $2 \log_{5/3} n$ phases.

Proof: Let $V(C)$ be the vertices of a connected component C. At the beginning of the algorithm, every vertex of $V(C)$ is the root of a star consisting of a single vertex. Each star is either the core of a star cluster or contained in a star cluster. The potential of each star that is the core of a cluster is 1, and the sum of the potential of the stars in clusters is at most the number of stars. Since the number of vertices in $V(C)$ is at most n, $\phi_1 \leq n$.

By Lemmas 4.6.7 and 4.6.8, $\phi_{k+2} \leq (3/5)\phi_k$, for every k. Thus, after every two phases, the potential of the forest in PG that contains the vertices of $V(C)$ decreases by a factor of at

least 5/3. Every vertex of $V(C)$ is contained in a single star when the potential is 1. Since $\phi_1 \leq n$, Algorithm II requires at most $2 \log_{5/3} n$ phases.

Theorem 4.6.10: Algorithm II requires O(log n) rounds.

Proof: The hooking and shortcut operations during each phase require O(1) rounds. With an **increment** primitive, the rolling synchronization for each phase requires one round. By Theorem 4.6.9, the number of phases required by Algorithm II is O(log n). Thus, the total number of rounds required by Algorithm II is O(log n). □

Algorithm II is much more efficient than a straightforward asynchronous implementation of the synchronous algorithm. If each iteration of a synchronous algorithm requires i PRAM steps, then each phase of a naive asynchronous algorithm would require i barrier synchronizations. The algorithm of Awerbuch and Shiloach (1987) requires $\log_{3/2} n$ iterations. Thus, the naive asynchronous algorithm would require a total of $i(\log_{3/2} n)$ barrier synchronizations.

Since our algorithm uses only three rolling synchronizations per phase, the total number of global synchronizations required by our algorithm is at most $6 \log_{5/3} n$. Thus the total number of global synchronizations is reduced by a factor of $0.210i$. This is a significant improvement since in each Part of the synchronous algorithm there is a test to determine whether a tree is a star. The Star_Check procedure alone requires at least 5 PRAM steps. Another improvement with our algorithm is that with rolling synchronization, after a processor completes a hooking or shortcut operation during the current phase, if the processor's endpoints have parents that are not the roots of trees, then the processor can make further progress via shortcut operations without waiting.

4.7. APRAM Connected Components Algorithm III

In Algorithm II, during each phase, Parts 1 and 3 are shortcut operations and Part 2 is a hooking operation. We can obtain a more efficient asynchronous algorithm by performing the hooking and shortcut operations in a different order depending on the phase. In Algorithm III, the order of operations depends on whether the phase is odd or even. A variable *phase* is used to store the phase of the algorithm. In Algorithm III, we will list only the operations.

APRAM Connected Components Algorithm III

Initialization

phase := 1;

while there are eligible pairs

  if (*phase* is odd) **then**

    Part 1: Shortcut operation;

    Part 2: Shortcut operation;

    rolling synchronization with counter;

    Part 3: Hooking operation;

    rolling synchronization with counter;

  else (* *phase* is even *)

    Part 1: Shortcut operation;

    rolling synchronization with counter;

    Part 2: Hooking operation;

    rolling synchronization with counter;

Part 3: Shortcut operation;

rolling synchronization with counter;

endif

phase := *phase* + 1;

endwhile

In the odd phases, only one rolling synchronization is needed after Part 2 to determine that every processor has performed at least two shortcut operations. We analyze the performance of Algorithm III in the same manner as for Algorithm II. The improvement in Algorithm III comes as a result of an improvement over Lemma 4.6.7.

Lemma 4.7.1: If k is odd, then the sum of the contributions of trees of F_k to trees of F_{k+2} is at most (4/7) ϕ_k.

Proof: The proof is similar to the proof of Lemma 4.6.7. Let T, $M'(T)$, T', $M''(T)$, T'', $M(T)$, and ρ be as in the proof of Lemma 4.6.7. We consider 6 cases. In each case, we show that $\rho \leq 4/7$.

Case 1: T is a star and $Cl(T)$ contains at least one star. The analysis is essentially the same as that for Cases 1 and 2 of Lemma 4.6.7. Thus, $\rho \leq 1/2$.

Case 2: T is a tall tree, $ht(T) \leq 4$, and $Cl(T)$ contains at least one star. Since k is odd, the algorithm performs at least two shortcut operations before the hooking operation in phase k. This case reduces to Case 1. Since T is a tall tree, $\Phi(T) = ht(T) \geq 2$. Since $Cl(T)$ contains at least one star, $\Phi(Cl(T), T) \geq 1$. At the end of phase k+1, $cont(Cl(T) + T, T'') = 1$. It follows that $\rho \leq 1/3$.

Case 3: T is a tall tree and $Cl(T)$ contains no stars. The analysis is the same as for Case 4 in the proof of Lemma 4.6.7. Thus, $\rho \leq 1/2$.

Case 4: The height of T is at least 5, T is grounded, and $Cl(T)$ contains at least one star. After at least two shortcut operations followed by a hooking operation in which T is grounded, at the end of phase k,

$$cont(M'(T)+T,T') \leq \lceil \lceil ht(T)/2 \rceil /2 \rceil + 1.$$

After at least one more shortcut operation followed by another hooking operation in which T' is grounded and at least one more shortcut operation, at the end of phase $k+1$,

$$\begin{aligned} cont(M(T)+T,T'') &\leq \lceil (\lceil (\lceil \lceil ht(T)/2 \rceil /2 \rceil + 1)/2 \rceil + 1)/2 \rceil \\ &= \lceil (\lceil (\lceil ht(T)/4 \rceil + 1)/2 \rceil + 1)/2 \rceil . \end{aligned} \tag{4.14}$$

If $5 \leq ht(T) \leq 20$, then by Inequality (4.14), $cont(M(T)+T,T'') \leq 2$. Since T is grounded, $cont(Stag(T),T'') \leq 1$. Since $ht(T) \geq 5$ and $\Phi(Cl(T),T) = 1$, it follows that $\rho \leq 1/2$.

Eliminating the ceiling operations in Inequality (4.14), we obtain $cont(M(T)+T,T'') \leq ht(T)/16 + 5/2$. Since T'' is grounded with respect to the stagnant stars of $Cl(T)$, if any, $cont(Stag(T),T'') \leq 1$. It follows that if $ht(T) \geq 21$, then since $\Phi(Cl(T),T) = 1$, $\rho \leq 7/32 < 4/7$.

Case 5: The height of T is at least 5, T is not grounded, and $Cl(T)$ contains at least one star. If $ht(T) \geq 5$, then stars of $Cl(T)$ may remain stagnant during phases k and $k+1$.

If $Cl(T)$ contains one star, then as in Case 6 of the proof of Lemma 4.6.7,

$$cont(M(T)+T,T'') \leq \lceil (\lceil ht(T)/8 \rceil + 1)/2 \rceil . \tag{4.15}$$

If $5 \le ht(T) \le 8$, then by Inequality (4.15), $cont(M(T)+T, T'') = 1$. Since $Cl(T)$ contains only one star, $cont(Stag(T), T'') \le 1$. Since $ht(T) \ge 5$ and $\Phi(Cl(T), T) = 1$, it follows that $\rho \le 1/3$.

Eliminating the ceiling operations in Inequality (4.15), we obtain $cont(M(T)+T, T'') \le ht(T)/16 + 2$. Since $cont(Stag(T), T'') \le 1$, it follows that if $ht(T) \ge 9$, then $\rho \le 57/160 < 4/7$.

If $Cl(T)$ contains at least two stars, then after at least two shortcut operations followed by a hooking operation in which T is not grounded, at the end of phase k,

$$cont(M'(T)+T, T') \le \lceil \lceil ht(T)/2 \rceil /2 \rceil + 2.$$

After at least one more shortcut operation followed by another hooking operation in which T' may not grounded and at least one more shortcut operation, at the end of phase k+1,

$$\begin{aligned} cont(M(T)+T, T'') &\le \lceil (\lceil (\lceil \lceil ht(T)/2 \rceil /2 \rceil + 2)/2 \rceil + 2)/2 \rceil \\ &= \lceil (\lceil ht(T)/8 \rceil + 1)/2 \rceil + 1. \end{aligned} \tag{4.16}$$

If $5 \le ht(T) \le 8$, then by Inequality (4.16), at the end of phase k+1, $cont(M(T)+T, T'') \le 2$. Since some stars of $Cl(T)$ may be stagnant and T'' might not be grounded with respect to the stars of $Stag(T)$, $cont(Stag(T), T'') \le 2$. It follows that if $ht(T) \ge 5$ and $\Phi(Cl(T), T) = 2$, then $\rho \le 4/7$.

Eliminating the ceiling operations in Inequality (4.16), we obtain $cont(M(T)+T, T'') \le ht(T)/16 + 3$. Since $cont(Stag(T), T'') \le 2$, it follows that if $ht(T) \ge 9$ and $\Phi(Cl(T), T) = 2$, then $\rho \le 89/176 < 4/7$.

Case 6: T is a free star. This is similar to Case 7 in the proof of Lemma 4.6.7. The analysis for this case is the same as the analysis for Case 7 of Lemma 4.6.7 with the exception of Subcase 7.3.2. Thus we consider only that subcase and show that $\rho' \leq 1/2 < 4/7$.

T is adopted by a star S in phase k. Let T_c be the tree for which $S \in Cl(T_c)$. In phase k, tree T_c adopts S. Let r_c be the root of T_c.

First suppose that T_c is grounded. After at least two shortcut operations followed by a hooking operation in which T_c, S, and T are merged in series, at the end of phase k,

$$cont(M'(T_c) + T_c + T, T') \leq \left\lceil \left\lceil ht(T_c)/2 \right\rceil /2 \right\rceil + 2.$$

If $ht(T_c) \leq 4$, then every star of $Cl(T_c)$ is adopted in phase k. Thus, after at least two more shortcut operations, at the end of phase $k+1$,

$$\begin{aligned} cont(M(T_c) + T_c + T, T'') &\leq \left\lceil \left\lceil (\left\lceil \left\lceil ht(T_c)/2 \right\rceil /2 \right\rceil + 2)/2 \right\rceil /2 \right\rceil \\ &= \left\lceil (\left\lceil ht(T_c)/8 \right\rceil + 1)/2 \right\rceil . \end{aligned} \tag{4.17}$$

If $ht(T_c) \leq 4$, then by Inequality (4.17), $cont(M(T_c) + T_c + T, T'') = 1$. Since every star of $Cl(T_c)$ is adopted in phase k, $cont(Stag(T_c), T'') = 0$. Since $ht(T_c) \geq 1$, $\Phi(Cl(T_c), T_c) = 1$, and $ht(T) = 1$, it follows that $\rho' \leq 1/3$.

If $ht(T_c) \geq 5$, then after at least one shortcut operation followed by a hooking operation in which T' is grounded and then at least one more shortcut operation, at the end of phase $k+1$,

$$\begin{aligned} cont(M(T_c) + T_c + T, T'') &\leq \left\lceil (\left\lceil (\left\lceil \left\lceil ht(T_c)/2 \right\rceil /2 \right\rceil + 2)/2 \right\rceil + 1)/2 \right\rceil \\ &= \left\lceil ht(T_c)/16 \right\rceil + 1. \end{aligned} \tag{4.18}$$

If $ht(T_c) \leq 8$, then as in Subcase 7.3.2, at the end of phase $k+1$, every star of $Cl(T_c)$ is adopted, and thus $cont(Stag(T_c), T'') = 0$. If $5 \leq ht(T_c) \leq 8$, then by Inequality (4.18), $cont(M(T_c) + T_c + T, T'') \leq 2$. Since $ht(T_c) \geq 5$, $\Phi(Cl(T_c), T_c) = 1$, and $ht(T) = 1$, it follows that $\rho' \leq 2/7$.

Eliminating the ceiling operations in Inequality (4.18), we obtain $cont(M(T_c) + T_c + T, T'') \leq ht(T_c)/16 + 2$. Since T_c is grounded, $cont(Stag(T_c), T'') \leq 1$. Since $\Phi(Cl(T_c), T_c) = 1$ and $ht(T) = 1$, it follows that if $ht(T_c) \geq 9$, then $\rho' \leq 57/176 < 1/2$.

Now suppose that T_c is not grounded. After at least two shortcut operations followed by a hooking operation in which T_c is not grounded, at the end of phase k,

$$cont(M'(T_c) + T_c + T, T') \leq \left\lceil \left\lceil ht(T_c)/2 \right\rceil /2 \right\rceil + 3.$$

If $ht(T_c) \leq 4$, then after the hooking operation in phase k, the parent of r_c is a vertex b, where $b \leq min\text{-}id(Cl(T_c))$. By Lemma 4.6.1, at the end of phase $k+1$, every star of $Cl(T_c)$ is adopted. Thus, $M(T_c) = Cl(T_c)$ and $cont(Stag(T_c), T'') = 0$. Since T_c adopts S and S adopts T during the hooking operation in phase k, after at least two shortcut operations, at the end of phase $k+1$, every vertex of T has a parent whose *id* is at most b. Thus, $cont(Cl(T_c) + T_c + T, T'') = 1$. Since $ht(T_c) \geq 1$, $\Phi(Cl(T_c), T_c) \geq 1$, and $ht(T) = 1$, it follows that $\rho' \leq 1/3$.

If $5 \leq ht(T_c) \leq 8$, then before the hooking operation in phase $k+1$, every vertex of T_c has a parent whose *id* is at most the *id* of r_c. Because T' incorporates T, if T' is merged only with stars of $Cl(T_c)$, then the height of the tallest tree that can be formed is at most $ht(T') + 1$. Thus, after at least one more shortcut operation followed by another hooking

operation in which T' may not be grounded and then at least one more shortcut operation, at the end of phase $k+1$,

$$cont(M(T_c)+T_c+T,T'') \le \left\lceil (\left\lceil (\left\lceil \left\lceil ht(T_c)/2 \right\rceil /2 \right\rceil + 3)/2 \right\rceil + 1)/2 \right\rceil \qquad (4.19)$$
$$= \left\lceil (\left\lceil (\left\lceil ht(T_c)/4 \right\rceil + 3)/2 \right\rceil + 1)/2 \right\rceil .$$

If $5 \le ht(T_c) \le 8$, then by Inequality (4.19), $cont(M(T_c)+T_c+T,T'') \le 2$. Since T_c is not grounded, $cont(Stag(T_c),T'') \le 2$. Since $ht(T_c) \ge 5$, $\Phi(Cl(T_c),T_c) = 2$, and $ht(T) = 1$, it follows that $\rho' \le 1/2$.

If $ht(T_c) \ge 8$, then after at least shortcut operation followed by another hooking operation in which T' may not be grounded and then at least one more shortcut operation, at the end of phase $k+1$,

$$cont(M(T_c)+T_c+T,T'') \le \left\lceil (\left\lceil (\left\lceil \left\lceil ht(T_c)/2 \right\rceil /2 \right\rceil + 3)/2 \right\rceil + 2)/2 \right\rceil \qquad (4.20)$$
$$= \left\lceil (\left\lceil ht(T_c)/4 \right\rceil + 3)/4 \right\rceil + 1.$$

Eliminating the ceiling operations in Inequality (4.20), we obtain $cont(M(T_c)+T_c+T,T'') \le ht(T_c)/16 + 3$. Since $cont(Stag(T_c),T'') \le 2$, $\Phi(Cl(T_c),T_c) = 2$, and $ht(T) = 1$, it follows that if $ht(T_c) \ge 9$, then $\rho' \le 89/192 < 1/2$. This completes the analysis for the subcase.

From these 6 cases, we conclude that the sum of the contributions of the trees of F_k to the trees of F_{k+2} is at most $(4/7)\,\phi_k$. □

The remainder of the analysis for Algorithm III is the same as for Algorithm II. As a result of Lemma 4.7.1, we conclude the following.

Theorem 4.7.2: Algorithm III requires at most $2 \log_{7/4} n$ phases.

Algorithm III reduces the number of rolling synchronizations that are required by Algorithm II in two ways. We have already shown that Algorithm III requires fewer phases. In addition, Algorithm III uses only five rolling synchronizations for every two consecutive phases whereas Algorithm II uses six. The total number of global synchronizations required by Algorithm III is at most $5 \log_{7/4} n$. Compared with the naive asynchronous algorithm, the number of global synchronizations is reduced by a factor of $0.276i$, and compared with Algorithm II, the number of global synchronizations is reduced by a factor of 1.31.

4.8. APRAM Spanning Forest Algorithms

4.8.1. Spanning forest algorithms

We can modify Algorithms II and III to compute a spanning forest of G by keeping track of the processors that merge trees. Each processor corresponds to an edge of G that joins two trees. Let r be the root of a tree T. If during a phase processor p hooks r to a tree that adopts T, then the edge of G corresponding to p is an edge of a spanning forest. Since p last updated $P(r)$, p is associated with r.

During a phase, T may be hooked more than once. In the asynchronous algorithm, $P(r)$ and the name of the processor that performed the hooking operation must be updated atomically. Otherwise, the algorithm might not find a spanning forest, as follows.

Suppose p is the processor that last hooked r. If $P(r)$ and the name of the processor that hooked r are updated separately, then since the algorithm is asynchronous, some processor q that hooks r before p may store its name after p does. The edge corresponding to q, however, may not be an edge between the trees merged by p. Note that in the synchronous algorithm the name of the processor is not stored until the trees have been

adopted. In the asynchronous algorithm, without synchronization, a processor does know whether it is the last processor to hook a tree.

During a hooking operation, the parent of r and the name of the processor that performs the hooking are updated simultaneously in $P(r)$ by incorporating the *id* of the parent and the name of the processor into a composite value. If processor p hooks r to a vertex a, then the algorithm sets $P(r) := a.p$. Initially, the algorithm sets $P(v) := v.0$ for each vertex $v \in V$.

If r is hooked to a vertex during a phase, then the last time r is hooked during the phase, T is adopted. The processor p that performs the last hooking operation corresponds to an edge of a spanning forest of G. Processor p does not hook any other vertices during the remainder of the algorithm since the endpoints of the edge assigned to p are contained in the same tree in PG. Also, after T is adopted, the name of the processor in $P(r)$ should not change. Thus, in the shortcut operations, only the *id* of the new parent of r is changed in $P(r)$.

After the vertices of each connected component are contained in a single star, if v is not the root of a tree, then the name of the processor in $P(v)$ corresponds to an edge of a spanning forest of G. Since each nonroot vertex is adopted exactly once, each nonroot vertex is associated with one edge of the spanning forest.

4.8.2. Minimum spanning forest algorithms

Suppose G is a graph with weighted edges, and without loss of generality, suppose the weights of the edges of G are unique. We give an APRAM algorithm for finding the minimum spanning forest (MSF) of G. The APRAM MSF algorithm may be obtained by modifying either Algorithm II or III.

To find the MSF of G, during each hooking operation, only the processor that corresponds to the edge of G of least weight leaving a partial component performs a hooking operation. To determine the edge of minimum weight, however, requires the MSF algorithm to first reduce each partial component to a star through shortcut operations.

Let T be a tree of PG with root r. The MSF algorithm maintains a variable $Min(r)$ to keep track of the edge of least weight that leaves T. Let e be an edge (u, v) of G such that u is a vertex of T and v is not. When u becomes a child of r through shortcut operations, then the processor corresponding to e executes a **replace-min** primitive on $Min(r)$ with the weight of e. When T becomes a star, then $Min(r)$ contains the weight of the edge of least weight leaving T. The processor corresponding to that edge then performs a hooking operation to merge T with another tree. Note that T cannot be hooked until it is a star. We keep track of the processors that perform the hooking operations through a variable $Proc(v)$ for each vertex v.

The root r of T has a variable $Count(r)$ to maintain a count of the number of children it has in PG. When a vertex v of T becomes a child of r through a shortcut operation, then the processor performing the shortcut operation atomically increments $Count(r)$ using the **increment** primitive. $Count(r)$ increases after each shortcut operation until T becomes a star. The algorithm determines that T is a star if after a shortcut operation $Count(r)$ does not increase. A barrier synchronization is required after each shortcut operation to ensure that every processor has had an opportunity to update $Count(r)$.

The analysis for the APRAM MSF Algorithm is similar to the analysis for Algorithms II and III. The main difference is that the stars of a cluster cannot be merged with the core of the cluster until the core has also become a star. The analysis is similar to the analysis of the

connected components algorithm except that no stars are merged with the core until the core is a star. It can be shown that after every two phases the potential of the forest in PG that contains the vertices that belong to a connected component of G decreases by at least a constant factor. Thus, the number of phases required by the algorithm is $O(\log n)$. Since each phase requires O(1) rounds, the total number of rounds needed to compute the MSF of G is $O(\log n)$.

4.9. Conclusions

We have presented three APRAM algorithms for finding the connected components of a graph. Algorithm I runs on a basic APRAM and requires $O(n \log n)$ rounds. Algorithms II and III run on an APRAM with **increment** and **replace-min** primitives and require $O(\log n)$ rounds. All three algorithms use $m + n$ processors.

Algorithm III requires fewer phases and global synchronizations than Algorithm II and is thus more efficient. One area of possible improvement is to find a more elaborate potential function and lower the bound on the number of phases required by Algorithms II and III. Another area of possible improvement is to reduce the number of global synchronizations that are required.

CHAPTER 5.

AN ASYNCHRONOUS PARALLEL ALGORITHM FOR COMPUTING BICONNECTED COMPONENTS

5.1. Introduction

Let $G = (V, E)$ be a connected graph with the set of vertices V and the set of edges E. Let $n = |V|$ and $m = |E|$. A vertex v is an *articulation point* if the removal of v and the edges incident on v leaves the remaining graph disconnected. A graph with no articulation points is *biconnected.* A maximal biconnected subgraph of G is a *biconnected component* of G.

Finding the biconnected components of a graph is a fundamental problem of graph theory. Some practical applications of finding the biconnected components of a graph include determining the reliability of communication and transportation networks.

Tarjan (1972) gave a sequential algorithm that uses depth-first search to find the articulation points and the biconnected components of a graph in $O(m+n)$ time. Savage and Ja'Ja' (1981) presented a parallel biconnected components algorithm for the CREW PRAM that runs in $O(\log^2 n)$ time using $O(\frac{n^3}{\log n})$ processors. Tsin and Chin (1984) presented an improved CREW PRAM algorithm that runs in $O(\log^2 n)$ time using $\frac{n^2}{\log^2 n}$ processors. Their algorithm is optimal for dense graphs.

Tarjan and Vishkin (1985) designed a biconnected components algorithm by reducing the biconnected components problem to a connected components problem. They presented an algorithm for the Arbitrary CRCW PRAM that runs in $O(\log n)$ time using $O(m+n)$

processors. They showed that their algorithm also runs on a CREW PRAM in $O(\log^2 n)$ time using the same number of processors. Then they gave an alternative implementation of their algorithm that runs on a CREW PRAM in $O(\frac{n^2}{p})$ time using any number $p \leq \frac{n^2}{\log^2 n}$ processors. Using the reduction idea of Tarjan and Vishkin, Cole and Vishkin (1986b) designed a biconnected components algorithm for the Arbitrary CRCW PRAM that runs in $O(\log n)$ time using $\frac{(m+n)\alpha(m,n)}{\log n}$ processors, where $\alpha(m,n)$ is the inverse Ackermann function.

In this chapter, we present an Asynchronous PRAM (APRAM) algorithm for computing the biconnected components of a graph. Various APRAM models have been defined by Gibbons (1989), Martel et al. (1989, 1990), Cole and Zajicek (1989, 1990), and Nishimura (1990). We use an APRAM with limited **read-modify-write** primitives (Wu, 1991). Our APRAM algorithm, which is based on the CRCW PRAM algorithm of Tarjan and Vishkin, requires $O(\log n)$ rounds using $O(m+n)$ processors.

In Section 5.2, we briefly review our APRAM model. In Section 5.3, we describe the synchronous biconnected components algorithm of Tarjan and Vishkin. In Section 5.4, we present our APRAM algorithm.

5.2. Model of Computation

The APRAM consists of a set of reliable processors, each with its own local memory, and a global shared memory through which processors communicate with one another. There is no global clock that synchronizes the processors, and access to the shared memory is also asynchronous. Each step of a processor of the APRAM consists of three stages. In the first stage, a processor may read a value from one shared memory cell into local memory. In the

second stage, a processor may perform a local computation on values in its local memory. In the third stage, a processor may write a value from local memory into one shared memory cell. At most one memory operation may be performed on a cell of the shared memory at any time; hence, each read operation and write operation is atomic. Thus, there is no ambiguity about the value that is read or written.

In addition, the APRAM has **read-modify-write** primitives. A processor executing a **read-modify-write** primitive atomically reads a value v from a cell c of the shared memory into local memory, performs a local computation that may depend on v or some other values stored in local memory, and then writes a value into cell c. The only **read-modify-write** primitives required by our algorithm are **replace-min** and **increment.**

An execution of an APRAM algorithm consists of a finite sequence of arbitrarily interleaved stages of the steps taken by all of the processors of the APRAM, with the restriction that when a processor executes a **read-modify-write** primitive, the stages of the step are atomic with respect to the memory cell that is accessed. We partition the execution of the algorithm into rounds, where a round is a minimal sequence of stages such that every processor takes at least one complete step. We measure the running time of our algorithm in rounds. For PRAM algorithms, the number of rounds is equal to the running time.

5.3. The Synchronous Biconnected Components Algorithm

Tarjan and Vishkin (1985) presented an efficient parallel algorithm for finding the biconnected components of a graph. Since our algorithm is based on their algorithm, we first describe their approach. Their main idea is to reduce the problem of finding the biconnected components of a graph G to finding the connected components of an auxiliary graph G',

where the connected components of G' correspond to the biconnected components of G. A similar approach was independently discovered by Tsin and Chin (1984). The auxiliary graph of Tsin and Chin contains many more edges, however, and thus their algorithm is less efficient.

Let e_1 and e_2 be two arbitrary edges of G. Define R to be a relation on the edges of G such that $e_1 R\ e_2$ if and only if either $e_1 = e_2$, or e_1 and e_2 are on a common simple cycle of G. Harary (1969) showed that R is an equivalence relation. The subgraphs of G induced by the equivalence classes of R are the biconnected components of G. An edge in a singleton equivalence class is a *bridge* of G. If e is an edge that is a bridge of G, then removing e disconnects G.

Tarjan and Vishkin define an auxiliary graph G' of G such that connected components of G' correspond to biconnected components of G. We describe how to derive G' from G. Let T be a rooted spanning tree of G. Let $v \rightarrow w$ denote an edge of T where v is the parent of w. Let $P(w)$ denote the parent of w in T. Number the vertices of T in preorder from 1 to n and let the number of each vertex be its *id*. We will refer to vertices by their *id*'s. The notation $u < v$ means that the *id* of vertex u is smaller than the *id* of vertex v.

The vertices of G' are the sets $\{u, v\}$ such that (u, v) is an edge of G. Each edge of G' has of one of the following forms:

Case 1: $(\{u, w\}, \{v, w\})$, where $u \rightarrow w$ is an edge of T and (v, w) is an edge of $G - T$ such that $v < w$.

Case 2: $(\{u, v\}, \{x, w\})$, where $u \rightarrow v$ and $x \rightarrow w$ are edges of T and (v, w) is an edge of $G - T$ such that neither v nor w is an ancestor of the other in T.

Case 3: $(\{u,v\}, \{v,w\})$, where $u \rightarrow v$ and $v \rightarrow w$ are edges of T and some edge of G joins a descendant of w in T with a vertex x such that x is not a descendant of v in T.

Tarjan and Vishkin proved the following theorem relating G and G'.

Theorem 5.3.1: (Tarjan and Vishkin, 1985) Two edges of G are in the same biconnected component of G if and only if their corresponding vertices are in the same connected component of G'.

Each edge e of $G - T$ defines a simple cycle in G consisting of e and the path in T joining the endpoints of e. Thus, all edges on this cycle belong to the same biconnected component. The edges of G' are such that the vertices of G' corresponding to the edges of a simple cycle in G are contained in the same connected component in G'.

We now outline the parts of the biconnected components algorithm of Tarjan and Vishkin. We then describe their parallel implementation.

Part 1: Find a spanning tree T of G. Number the vertices of T from 1 to n in preorder. Compute the number of descendants $nd(v)$ of each vertex v.

Part 2: For each vertex v, compute $low(v)$ (resp. $high(v)$), the lowest (highest) vertex that is either a descendant of v or adjacent to a descendant of v by an edge of $G - T$.

Part 3: Construct the graph G'', the subgraph of G' induced by the edges of T, as follows. For each edge (w,v) in $G - T$ such that $v + nd(v) \leq w$, add the edge $(\{P(v),v\}, \{P(w),w\})$ to G''. These are the edges of Case 2. For each edge of $v \rightarrow w$ of T such that $v \neq 1$, add the edge $(\{P(v),v\}, \{v,w\})$ to G'' if $low(w) < v$ or $high(w) \geq v + nd(v)$. These are the edges of Case 3.

Part 4: Find the connected components of G''.

Part 5: Extend the equivalence relation R on the edges of T to the edges of $G - T$ by defining edge (v, w) equivalent to edge $(P(w), w)$ for each edge (v, w) of $G - T$ such that $v < w$. These are the edges of Case 1.

The parallel implementation of the algorithm uses an Arbitrary CRCW PRAM. Each edge of G is represented by two oppositely directed edges. The algorithm assigns a processor $p(i,j)$ to each edge (i,j) and a processor $p(v)$ to each vertex v. Thus, the number of processors required by the algorithm is at most $2m + n$.

In Part 1, the algorithm first constructs a spanning tree of G using a modification of the connected components algorithm of Shiloach and Vishkin (1982). This takes $O(\log n)$ time using $O(m + n)$ processors.

Next, the algorithm constructs a circular list of the directed edges of the tree to form an Eulerian tour of the tree. The Eulerian tour corresponds to the order and direction in which edges are traversed during a depth-first search traversal of the tree starting from an arbitrary vertex. The tree is rooted by breaking the Eulerian tour at an arbitrary edge. Tarjan and Vishkin call this the *Euler tour technique on trees.*

They combine the Euler tour technique with the doubling technique for list ranking (Fortune and Wyllie, 1978) and show that several tree functions, including computing the preorder and postorder numbers of vertices and the number of descendants for all vertices of the tree, can be performed in $O(\log n)$ time using $O(n)$ processors. We next describe the doubling technique.

Let L be a list of n elements. For each element k in L, let $next(k)$ be a pointer to the next element in the list. The last element in the list points to itself. The *rank* of an element is

its distance from the end of the list; the rank of the last element is 0. Let $rank(k)$ be the rank of element k. The rank of each element can be determined with the following list ranking algorithm. A processor $p(k)$ is assigned to each element k. Each processor $p(k)$ maintains a pointer $N(k)$ that initially points to the element $next(k)$. During the execution of the algorithm, $N(k)$ progressively points to elements closer to the end of the list. When the algorithm terminates, $N(k)$ for every element k points to the last element in the list.

List Ranking Algorithm

For each processor $p(k)$

$N(k) := next(k)$;

if $N(k) \neq next(k)$ **then**
    $dist(k) := 1$;
else
    $dist(k) := 0$;
endif

repeat $\lceil \log n \rceil$ **times**

    if $N(k) \neq N(N(k))$ **then**
        $dist(k) := dist(k) + dist(N(k))$;
        $N(k) := N(N(k))$;
    endif

end repeat

$rank(k) := dist(k)$;

Pointer $N(k)$ is recursively "doubled" by updating $N(k)$ to $N(N(k))$. After i iterations, for each element k whose rank is at most 2^i, $N(k)$ points to the element at the end of the list. Thus, the rank of each element can be determined after $O(\log n)$ iterations. We now return to the discussion of the parallel algorithm of Tarjan and Vishkin.

In Part 2, the algorithm computes $low(v)$ and $high(v)$ for each vertex v. We sketch their method. We will describe the computation of $low(v)$; the computation of $high(v)$ is similar. Each vertex v has an adjacency list of the vertices adjacent to v. Let $local\_low(v)$ be the vertex u with the lowest preorder number such that u is either v or a neighbor of v such that (u,v) is an edge of $G-T$. For each vertex v, $local\_low(v)$ is computed by applying the doubling technique on the adjacency list of v. This requires $O(\log n)$ time using $O(m)$ processors.

For an interval $[i,j]$, let $global\_low[i,j] = \min\{local\_low(x) \mid i \le x \le j\}$. For each α, $0 \le \alpha \le \log n$, and for each x, $1 \le x \le n/2^{\alpha}$, compute $global\_low[(x-1)2^{\alpha}+1, x\,2^{\alpha}]$. The total number of *global_low* values is $O(n)$. The *global_low* values can be computed in $O(\log n)$ time using n processors. In the first iteration, compute $global\_low[i,i]$ for each i, $1 \le i \le n$, in parallel. This takes O(1) time using n processors. Then in each successive iteration, compute the *global_low* values for intervals that are twice the size of the intervals of the previous iteration by finding the smaller of two *global_low* values for two consecutive intervals. Finally, compute $low(v)$ for each vertex v using the formula

$$low(v) = \min\{local\_low(x) \mid v \le x \le v + nd(v) - 1\}.$$

The values of x are the preorder numbers of the vertices that are descendants of v. Any interval $[i,j]$, $1 \le i \le j \le n$, can be represented as a union of at most $2\log n$ *global_low* values. Thus, for each vertex v, the minimum value of $local\_low(x)$ in the interval $[v, v + nd(v) - 1]$ can be determined by checking at most $2\log n$ *global_low* values. Thus, $low(v)$ for each vertex v can be determined in $O(\log n)$ time using n processors. We conclude that the entire computation of Part 2 requires $O(\log n)$ time using $O(n)$ processors.

In Part 3, the algorithm constructs the auxiliary graph G''. Since T has $n-1$ edges, G'' has $n-1$ vertices. Part 3 requires O(1) time using O(m) processors since testing the conditions for each possible edge of G'' takes O(1) time. The graph G'' has at most $m-1$ edges.

In Part 4, the algorithm finds the connected components of G''. The algorithm of Shiloach and Vishkin requires O($\log n$) time using O($m+n$) processors. The vertices of G'' that belong to the same connected component correspond to the edges of G in the same equivalence class and thus the same biconnected component.

In Part 5, the algorithm extends the equivalence relation R found in Part 4 to the edges of $G-T$. If (i,j) is a nontree edge such that $i<j$, then (i,j) is assigned to the same connected component as edge $(P(j),j)$. Part 5 takes O(1) time using O(m) processors.

Arbitrary concurrent writing arises in the connected components algorithm of Shiloach and Vishkin, and thus is required in Parts 1 and 4 of the biconnected components algorithm. Since each Part of the algorithm runs in O($\log n$) time and uses O($m+n$) processors, the biconnected components algorithm of Tarjan and Vishkin runs in O($\log n$) time using O($m+n$) processors. Two other problems that can be solved by variants of the biconnected components algorithm in the same resource bounds are finding all of the bridges of a graph, and directing the edges of a bridgeless graph so that the resulting directed graph is strongly connected.

5.4. APRAM Biconnected Components Algorithm

We now present our APRAM biconnected components algorithm. Our algorithm is essentially an asynchronous implementation of the biconnected components algorithm of

Tarjan and Vishkin. We describe our implementation of the various parts of the synchronous algorithm.

In Part 1, we can find a spanning tree T of G using a modification of APRAM connected components algorithm given in Chapter 4. Part 1 requires $O(\log n)$ rounds using $m+n$ processors.

Next, we number the vertices of T in preorder and compute the number of descendants of each vertex using the Euler tour technique on trees and the doubling technique. The asynchronous implementation of the doubling technique, however, requires a barrier synchronization between the read and write stages of each doubling operation. This ensures that the updates of the $dist(k)$ and $N(k)$ variables for each element k are coordinated. With an **increment** primitive, each barrier synchronization requires one round. Since the Euler tour technique on trees requires $O(\log n)$ iterations using $O(n)$ processors, the number of rounds required $O(\log n)$.

We show that if barrier synchronization is not used during the doubling operation, then the computed distance to the end of the list may be too small. For an element k, suppose that at the time $dist(k)$ is updated, $N(k) := k'$ and $N(N(k)) := k''$. Then, when $dist(k)$ is updated, $dist(k) := dist(k) + dist(k')$. Meanwhile, since the processors are asynchronous, while $dist(k)$ is being updated, $N(k')$ may also be updated. Subsequently, when $N(k)$ is updated, $N(k)$ does not point to element k'', but rather to an element closer to the end of the list. Thus, the computed distance to the end of the list will be smaller than the actual distance.

In Part 2, the algorithm asynchronously computes $low(v)$ and $high(v)$ using the same method as in the synchronous algorithm. Again, barrier synchronization is needed during the doubling operations. Part 2 requires $O(\log n)$ rounds using $O(n)$ processors.

In Part 3, the algorithm asynchronously constructs the auxiliary graph G''. Part 3 takes O(1) rounds using O(m) processors.

In Part 4, the algorithm asynchronously finds the connected components of G'' using the APRAM connected components algorithm of Chapter 4. Part 4 requires O($\log n$) rounds using $m + n$ processors.

In Part 5, the algorithm extends the equivalence relation R found in Part 4 to the edges of $G - T$. Part 5 takes O(1) rounds using O(m) processors.

In addition, a barrier synchronization is inserted between each of the Parts so that processors can determine when a part is completed. Since each of the Parts of the asynchronous algorithm requires O($\log n$) rounds and uses O($m + n$) processors, and each barrier synchronization requires one round, our APRAM biconnected components algorithm requires O($\log n$) rounds using O($m + n$) processors.

REFERENCES

Aho, A. V., Ullman, J. D., Wyner, A. D., and Yannakakis, M. (1982), Bounds on the Size and Transmission Rate of Communications Protocols, *Computers & Mathematics with Applications,* **8,** 205-214.

Akl, S. (1989), *The Design and Analysis of Parallel Algorithms,* Prentice Hall, Englewood Cliffs, NJ.

Anderson, R. J., and Woll, H. S. (1991), Wait-Free Parallel Algorithms for the Union-Find Problem, Technical Report 91-04-05, University of Washington.

Aspnes, J. and Herlihy, M. (1990), Wait-Free Data Structures in the Asynchronous PRAM Model, *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures,* 340-349.

Awerbuch, B. (1987), Optimal Distributed Algorithms for Minimum Weight Spanning Tree, Counting, Leader Election and Related Problems, *Proceedings of the 19th Annual ACM Symposium on Theory of Computing,* 230-240.

Awerbuch, B. and Shiloach, Y. (1987), New Connectivity and MSF Algorithms for Shuffle-Exchange Network and PRAM, *IEEE Transactions on Computers,* **36,** 1258-1263.

Axelrod, T. S. (1986), Effects of Synchronization Barriers on Multiprocessor Performance, *Parallel Computing,* **3,** 129-140.

Bartlett, K. A., Scantlebury, R. A., and Wilkinson, P. T. (1969), A Note on Reliable Full-Duplex Transmission over Half-Duplex Links, *Communications of the ACM,* **12,** 260-261.

Batcher, K. (1968), Sorting Networks and Their Applications, *Proceedings of the American Federation of Information Processing Societies Conference,* **32,** 307-314.

Boppana, R. B. (1989), Optimal Separations Between Concurrent-Write Parallel Machines, *Proceedings of the 21st Annual ACM Symposium on Theory of Computing,* 320-326.

Brand, D. and Zafiropulo, P. (1983), On Communicating Finite-State Machines, *Journal of the Association for Computing Machinery,* **30,** 323-342.

Cheriton, D. and Tarjan, R. E. (1976), Finding Minimum Spanning Trees, *SIAM Journal of Computing,* **5,** 724-742.

Chin, F. Y., Lam, J., and Chen, I. (1982), Efficient Parallel Algorithms for some Graph Problems, *Communications of the ACM,* **25,** 659-665.

Chlebus, B. S., Diks, K., Hagerup, T., and Radzik, T. (1988), Efficient Simulations Between Concurrent-Read Concurrent-Write PRAM Models, *Lecture Notes in Computer Science,* **324,** 231-239.

Chor, B., Israeli, A., and Li, M. (1987), On Processor Coordination Using Asynchronous Hardware, *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing,* 169-178.

Cole, R. (1988), Parallel Merge Sort, *SIAM Journal of Computing,* **17,** 770-785.

Cole, R. and Vishkin, U. (1986a), Deterministic Coin Tossing and Accelerating Cascades: Micro and Macro Techniques for Designing Parallel Algorithms, *Proceedings of the 18th Annual ACM Symposium on Theory of Computing,* 206-219.

Cole, R. and Vishkin, U. (1986b), Approximate and Exact Parallel Scheduling with Applications to List, Tree, and Graph Problems, *Proceedings of the 27th Annual Symposium on Foundations of Computer Science,* 478-491.

Cole, R. and Zajicek, O. (1989), The APRAM: Incorporating Asynchrony into the PRAM Model, *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures,* 169-178.

Cole, R. and Zajicek, O. (1990), The Expected Advantage of Asynchrony, *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures,* 85-94.

Danthine, A. (1982), Protocol Representation with Finite State Models, *Computer Network Architectures and Protocols,* Ed. Paul E. Green, Jr., 579-606.

Dubois, M. and Briggs, F. A. (1991), The Run-Time Efficiency of Parallel Asynchronous Algorithms, *IEEE Transactions on Computers,* **40,** 1260-1266.

Fich, F. E., Ragde, P., and Wigderson, A. (1988a), Relations Between Concurrent-Write Models of Parallel Computation, *SIAM Journal on Computing,* **17,** 606-627.

Fich, F. E., Ragde, P., and Wigderson, A. (1988b), Simulations Among Concurrent-Write PRAMs, *Algorithmica,* **3,** 43-51.

Fischer, M. J., Lynch, N. A., and Paterson, M. S. (1985), Impossibility of Distributed Consensus with One Faulty Process, *Journal of the Association for Computing Machinery,* **32,** 374-382.

Fortune, S. and Wyllie, J. (1978), Parallelism in Random Access Machines, *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*, 114-118.

Fredman, M. L. and Tarjan, R. E. (1987), Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms, *Journal of the Association for Computing Machinery*, **34**, 596-615.

Gallager, R., Humblet, P., and Spira, P. (1983), A Distributed Algorithm for Minimum-Weight Spanning Trees, *ACM Transactions on Programming Languages and Systems*, **5**, 66-77.

Gazit, H. (1991), An Optimal Randomized Parallel Algorithm for Finding Connected Components in a Graph, *SIAM Journal of Computing*, **20**, 1046-1067.

Gibbons, A. and Rytter, W. (1988), *Efficient Parallel Algorithms*, Cambridge University Press, Cambridge, Great Britain.

Gibbons, P. B. (1989), A More Practical PRAM Model, *Proceedings of the 1989 ACM Symposium on Parallel Algorithms and Architectures*, 158-168.

Gouda, M. G. (1985), On 'A Simple Protocol Whose Proof Isn't': The State Machine Approach, *IEEE Transactions on Communications*, **33**, 382-384.

Gouda, M. G. and The, K.-S. (1985), Modeling Physical Layer Protocols Using Communicating Finite State Machines, *Proceedings of the 9th Data Communications Symposium*, 54-62.

Graham, R. L. and Hell, P. (1985), On the History of the Minimum Spanning Tree Problem, *Annals of the History of Computing*, **7**, 43-57.

Hailpern, B. (1985), A Simple Protocol Whose Proof Isn't, *IEEE Transactions on Communications*, **33**, 330-337.

Halpern, J. Y. and Zuck, L. D. (1987), A Little Knowledge Goes a Long Way: Simple Knowledge-Based Derivations and Correctness Proofs for a Family of Protocols, *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, 269-280.

Han, Y. and Wagner, R. (1990), An Efficient and Fast Parallel Connected Component Algorithm, *Journal of the Association for Computing Machinery*, **37**, 626-642.

Harary, F. (1969), *Graph Theory*, Addison-Wesley, Reading, MA.

Herlihy, M. (1988), Impossibility and Universality Results for Wait-Free Synchronization, *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing,* 276-290.

Hirschberg, D. S. (1982), Parallel Graph Algorithms Without Memory Conflicts, *Proceedings of the 20th Annual Allerton Conference on Communications, Control, and Computing,* 257-263.

Hirschberg, D. S., Chandra, A. K., and Sarwate, D. V. (1979), Computing Connected Components on Parallel Computers, *Communications of the ACM,* **22,** 461-464.

Ja'Ja', J. (1991), *An Introduction to Parallel Algorithms,* Addison-Wesley, Reading, MA.

Johnson, D. B. and Metaxas, P. (1991), Connected Components in $O(\lg^{3/2}|V|)$ Parallel Time for the CREW PRAM, *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science,* 688-697.

Karp, R. M. and Ramachandran, V. (1990), Parallel Algorithms for Shared-Memory Machines, *Handbook of Theoretical Computer Science, Vol. A: Algorithms and Complexity,* Ed. J. van Leeuwen, Elsevier (Amsterdam) and MIT Press (Cambridge, Mass.), 869-941.

Kruskal, C. P., Rudolph, L., and Snir M. (1990), Efficient Parallel Algorithms for Graph Problems, *Algorithmica,* **5,** 43-64.

Lamport, L. (1986), On Interprocess Communication, Part II: Algorithms, *Distributed Computing,* **1,** 86-101.

Loui, M. C. and Abu-Amara, H. H. (1987), Memory Requirements for Agreement Among Unreliable Asynchronous Processes, *Advances in Computing Research,* **4,** JAI Press, Inc., 163-183.

Lynch, N., Mansour, Y., and Fekete, A. (1988), The Data Link Layer: Two Impossibility Results, *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing,* 149-170.

Mano, M. M. (1982), *Computer System Architecture,* Prentice-Hall, Inc., Englewood Cliffs, NJ 07632.

Martel, C., Park, A., and Subramonian, R. (1989), Optimal Asynchronous Algorithms for Shared Memory Parallel Computers, Report CSE-89-8, Division of Computer Science, University of California, Davis.

Martel, C., Subramonian, R., and Park, A. (1990), Asynchronous PRAMs are (almost) as good as Synchronous PRAMs, *Proceedings of the 30th Annual Symposium on Foundations of Computer Science,* 590-599.

Nishimura, N. (1990), Asynchronous Shared Memory Parallel Computation, *Proceedings of the 2nd Annual ACM Symposium on Parallel Algorithms and Architectures,* 76-84.

Peng, W. and Purushothaman, S. (1989), Towards Dataflow Analysis of Communicating Finite State Machines, *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing,* 45-58.

Savage, C. and Ja'Ja', J. (1981), Fast, Efficient Parallel Algorithms for Some Graph Problems, *SIAM Journal of Computing,* **10,** 682-691.

Shiloach, Y. and Vishkin, U. (1982), An O(n log n) Parallel Connectivity Algorithm, *Journal of Algorithms,* **3,** 57-67.

Tanenbaum, A. S. (1981), *Computer Networks,* Prentice Hall, Inc., Englewood Cliffs, NJ 07632.

Tarjan, R. E. (1972), Depth-First Search and Linear Graph Algorithms, *SIAM Journal of Computing,* **1,** 146-160.

Tarjan, R. E. and Vishkin, U. (1985), An Efficient Parallel Biconnectivity Algorithm, *SIAM Journal of Computing,* **14,** 862-874.

Tempero, E. and Ladner, R. E. (1990), Tight Bounds for Weakly-Bounded Protocols, *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing,* 205-218.

Tsin, Y. H. and Chin, F. Y. (1984), Efficient Parallel Algorithms for a Class of Graph Theoretic Problems, *SIAM Journal of Computing,* **13,** 580-599.

Wang, D. and Zuck, L. D. (1989), Tight Bounds for the Sequence Transmission Problem, *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing,* 73-83.

Wu, M. M. (1990), An O(log n) Time Common CRCW PRAM Algorithm for Minimum Spanning Tree, Technical Report UILU-ENG-90-2250 (ACT-114), Coordinated Science Laboratory, University of Illinois at Urbana-Champaign.

Wu, M. M. (1991), Asynchronous PRAM Algorithms for Graph Connectivity and Related Problems, in preparation.

Wu, M. M. and Loui, M. C. (1991), Modeling Robust Asynchronous Communication Protcols with Finite-State Machines, to appear in *IEEE Transactions on Communications*.

Yao, A. C. (1975), An O(|E| log log |V|) Algorithm for Finding Minimum Spanning Trees, *Information Processing Letters*, 4 21-23.

Yu, Y. and Gouda, M. G. (1982), Deadlock Detection for a Class of Communicating Finite State Machines, *IEEE Transactions on Communications*, **30**, 2514-2518.

Zafiropulo, P., West, C. H., Rudin, H., Cowan, D. D., and Brand, D. (1980), Towards Analyzing and Synthesizing Protocols, *IEEE Transactions on Communications*, **28**, 651-661.

VITA

Michael M. Wu was born on January 23, 1964, in Oak Park, Illinois. He received a B.S. degree in Electrical Engineering with Honors from the University of Illinois at Urbana-Champaign in 1985. He then entered graduate school at the University of Illinois and received the M.S. and Ph.D. degrees in Electrical Engineering in 1987 and 1992, respectively. His thesis research was directed by Dr. Michael C. Loui.

Dr. Wu is a member of Tau Beta Pi and Eta Kappa Nu and an associate member of Sigma Xi. He is also a member of the Institute of Electrical and Electronics Engineers and the Association for Computing Machinery. His research interests include the design of parallel and distributed computer systems. His publications include:

An Efficient Distributed Algorithm for Maximum Matching in General Graphs, by M. M. Wu and M. C. Loui, *Algorithmica*, **5** (1990), 383-406.

Modeling Robust Asynchronous Communication Protocols with Finite-State Machines, by M. M. Wu and M. C. Loui, accepted for publication in *IEEE Transactions on Communications*.

An O(log n) Time Common CRCW PRAM Algorithm for Minimum Spanning Tree, Technical Report ACT-114, Coordinated Science Laboratory, University of Illinois at Urbana-Champaign, 1990.

Asynchronous PRAM Algorithms for Graph Connectivity and Related Problems, submitted for publication.